ABSTRACT

Title of Document:	CYBERSECURITY FOR INTELLECTUAL PROPERTY: DEVELOPING PRACTICAL FINGERPRINTING TECHNIQUES FOR INTEGRATED CIRCUITRY
	Carson Dunbar, Doctor of Philosophy, 2015
Directed By:	Professor Gang Qu, Department of Electrical and Computer Engineering

The system on a chip (SoC) paradigm for computing has become more prevalent in modern society. Because of this, reuse of different functional integrated circuits (ICs), with standardized inputs and outputs, make designing SoC systems easier. As a result, the theft of intellectual property for different ICs has become a highly profitable business. One method of theft-prevention is to add a signature, or fingerprint, to ICs so that they may be tracked after they are sold. The contribution of this dissertation is the creation and simulation of three new fingerprinting methods that can be implemented automatically during the design process. In addition, because manufacturing and design costs are significant, three of the fingerprinting

methods presented, attempt to alleviate costs by determining the fingerprint in the post-silicon stage of the VLSI design cycle.

Our first two approaches to fingerprint ICs, are to use Observability Don't Cares (ODCs) and Satisfiability Don't Cares (SDCs), which are almost always present in ICs, to hide our fingerprint. ODCs cause an IC to ignore certain internal signals, which we can utilize to create fingerprints that have a minimal performance overhead. Using a heuristic approach, we are also able to choose the overhead the gate will have by removing some fingerprint locations. The experiments show that this work is effective and can provide a large number of fingerprints for more substantial circuits, with a minimal overhead. SDCs are similar to ODCs except that they focus on input patterns, to gates, that cannot exist. For this work, we found a way to quickly locate most of the SDCs in a circuit and depending on the input patterns that we know will not occur, replace the gates to create a fingerprint with a minimal overhead. We also created two methods to implement this SDC fingerprinting method, each with their own advantages and disadvantages. Both the ODC and SDC fingerprinting methods can be implemented in the circuit design or physical design of the IC, and finalized in the post-silicon phase, thus reducing the cost of manufacturing several different circuits.

The third method developed for this dissertation was based on our previous work on finite state machine (FSM) protection to generate a fingerprint. We show that we can edit ICs with incomplete FSMs by adding additional transitions from the set of don't care transitions. Although the best candidates for this method are those with unused states and transitions, additional states can be added to the circuit to generate

additional don't care transitions and states, useful for generating more fingerprints. This method has the potential for an astronomical number of fingerprints, but the generated fingerprints need to be filtered for designs that have an acceptable design overhead in comparison to the original circuit.

Our fourth and final method for IC fingerprinting utilizes scan-chains which help to monitor the internal state of a sequential circuit. By modifying the interconnects between flip flops in a scan chain we can create unique fingerprints that are easy to detect by the user. These modifications are done after the design for test and during the fabrication stage, which helps reduce redesign overhead. These changes can also be finalized in the post-silicon stage, similar to the work for the ODC and SDC fingerprinting, to minimize manufacturing costs.

The hope with this dissertation is to demonstrate that these methods for generating fingerprints, for ICs, will improve upon the current state of the art. First, these methods will create a significant number of unique fingerprints. Second, they will create fingerprints that have an acceptable overhead and are easy to detect by the developer and are harder to detect or remove by the adversary. Finally, we show that three of the methods will reduce the cost of manufacturing by being able to be implemented in the later stages of their design cycle.

CYBERSECURITY FOR INTELLECTUAL PROPERTY: DEVELOPING PRACTICAL FINGERPRINTING TECHNIQUES FOR INTEGRATED CIRCUITRY

By

Carson J. Dunbar

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2015

Advisory Committee:
Professor Gang Qu, Chair
Professor Min Wu
Assistant Professor Dana Dachman-Soled
Assistant Professor Charalampos Papamanthou
Professor Michael Fu

DEDICATION

This dissertation is dedicated to my parents Carson and Carol Dunbar, and my brother Alex Dunbar who have always believed in me and continue to push me to be the best I can be.

ACKNOWLEDGEMENTS

I would like to acknowledge my adviser, Dr. Gang Qu, for all of his hard work helping me get through my PhD program and introducing me to the world of hardware security. Without his understanding and patience, I'm not sure if I would have finished.

I would also like to acknowledge Dr. Aijiao Cui who ran experiments and gathered results for the last topic of this research, the Scan Chain Fingerprint.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION AND MOTIVATION

A. Introduction

In recent years, the system on a chip (SoC) paradigm has increased in popularity due to its modular nature. System designers can pick integrated circuits (ICs), considered as intellectual property (IP), that are produced for specific functionality and fit them together to achieve a specific goal. This leads to a culture of reuse based design [1]. As a result, IP theft has become profitable as well as a threat to IP developers, vendors, and the SoC industry in general, which motivates the IP protection problem [1, 2, 3, 4].

As recently as 2011, it was estimated that counterfeit circuits make up approximately 1% of the market with a financial loss of approximately $100 billion worldwide [5]. In addition, the number of reported counterfeiting incidents quadrupled between 2009 and 2011 [6]. Many of these counterfeit devices find their way into mission critical devices for the military and aerospace. These ICs can be of poorer quality and fail quicker than brand new devices from a trusted seller, as seen in Figure 1. A quicker fail time may lead to a number of ill-effects as specified in Table 1. As a result it can be understood that there is a need for a way to fight IC piracy.

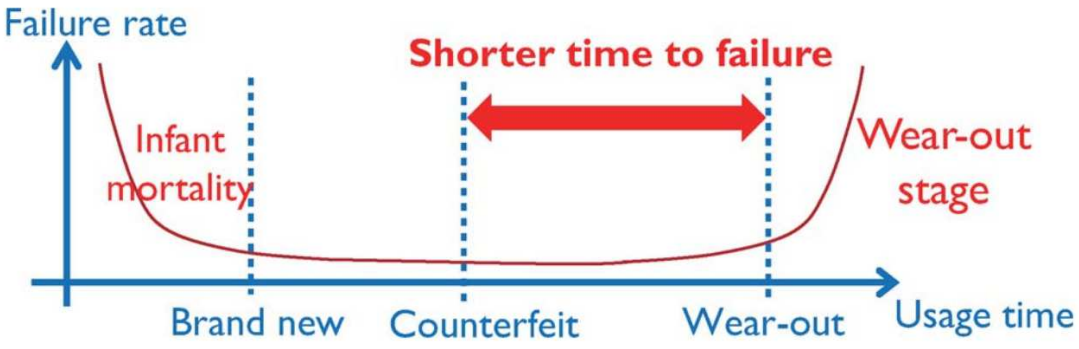


Figure 1. Classical bathtub curve, illustrating typical device failure characteristics[5]

Table 1. Possible Effects of Counterfeit ICs [7]

Government	Industry	Consumer
National security or civilian safety issues	Costs to mitigate this risk	Costs when products fail due to lower quality and reliability of counterfeit parts
Costs of enforcement	Costs to replace failed parts	
Lost tax revenue due to illegal sales of counterfeit parts	Lost sales	Potential safety concerns
	Lost brand value or damage to business image	

Traditionally, the normal very-large-scale integration (VLSI) design cycle, which develops IP cores, has a number of steps as seen in Figure 2. When early groups would develop ICs, it would be done in-house where they had their own designers and foundries and the process was strictly controlled. Now it is easier, and more cost efficient, to split these steps up among groups that specialize in different aspects of VLSI design. As a simple example, one may design the system specification and hand that data to another party who designs everything from the architecture to the

physical layout of the device. This layout could be given back to the person who created the system specification who in turn, gives it to a foundry who creates it using whatever process technology they have.

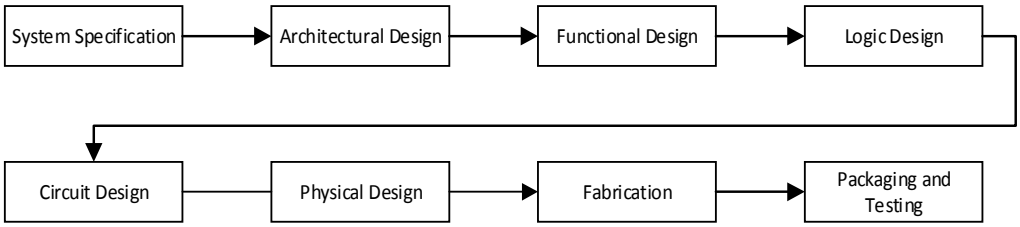


Figure 2. Simple VLSI Design Cycle [8]

At this point two new parties have had access to the IP of the creator. The addition of these third parties to the VLSI design cycle creates a substantial security risk and although the simple example above only uses two additional parties, many more could be added. Logically then, for every party involved, the chance of malicious behavior increases.

One risk of the multi-party design cycle is the addition of Trojan Horses, or simply Trojans, to the design. Trojans can be simple changes to the circuit level design or significant changes to the functional or logical design, with the intention of damaging the circuit, stopping functionality at critical points, or siphoning off data that was meant to be secure.

In addition to the multi-party design cycle issue, the entities that purchase or lease an IP introduce risk as well. Once a design is completed, it can be considered an IP core, and at that point it is vulnerable to theft by duplication. If a group does all of the design work themselves, or with a trusted third party, that the weak points for duplication occur once a client leases a design or buys the design or the physical IC or

the IP design is sent to a third party for fabrication. At each point, the third party can simply copy the physical layout of the device and claim it as their own.

B. Motivation

Watermarking and fingerprinting for circuits are two important methods used to protect IPs from unwarranted duplication. They have been used for a variety of IPs such as physical documents and digital media [9] with varied success. In ICs, watermarking methods attempt to either add functionality to a circuit or piggy-back on current functionality in order to create a code that is nearly impossible to remove. This code allows someone to prove that the circuit in question is their own.

Fingerprinting ICs is a slightly more relaxed idea in comparison to watermarking. It attempts to either use a circuit's properties or additional properties that the designer adds in to help identify the source of the circuit. The fingerprint may not be strong enough to prove a case in court, but it allows a designer to determine untrustworthy customers. It is also harder to trace and break as there may be hundreds of changes to a circuit and even if several adversaries collude, it would be difficult to find all fingerprint locations. As a quick motivational example, Figure 3 shows what a fingerprint may look like in an IC. Each location circled in red would be a location where a small modification could be implemented. The easy to read identifier would be the fingerprint bit string where a 0 represents a modification that was not used and a 1 would be a modification that was used.

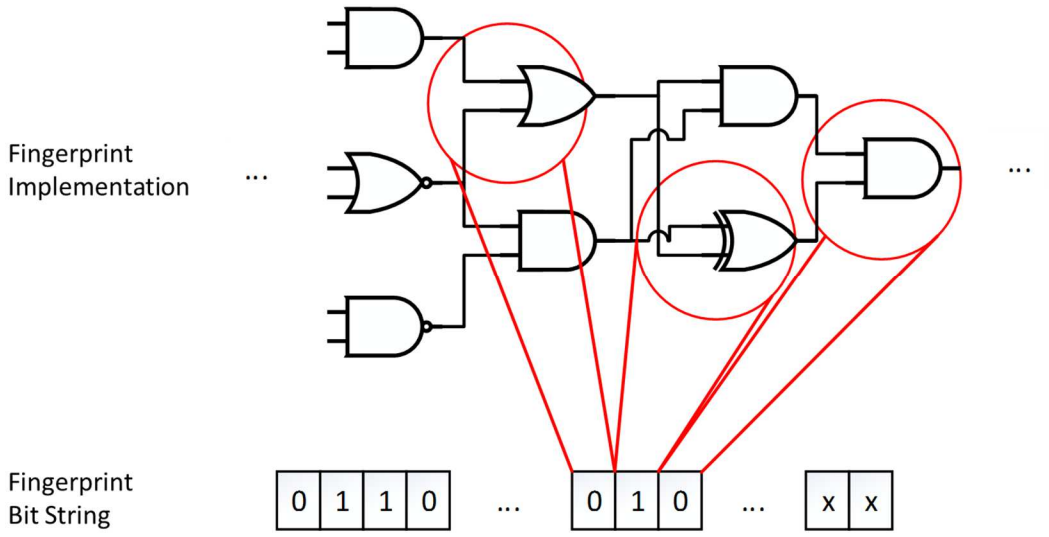


Figure 3. Simple fingerprint example

Currently there are only two other major works in regards to fingerprints for ICs. There are many techniques for circuit manipulation that result in only a small performance overhead. The goal of this work was to use some to see if it was possible to generate a large number of unique fingerprints. The secondary goal was to generate fingerprints that could be implemented in post-production because all current methods would require a new manufacturing process for each individual circuit, which would be very costly. These post-production fingerprints would be manufactured with "flexibilities" that could be hardened in the post-silicon stage, through the use of things like fuses, or other post-silicon modification techniques.

C. Dissertation Outline

This dissertation shows the work done to construct several procedures for creating practical fingerprints for ICs, some of which that can be implemented in later steps of the design cycle, thus reducing their cost. Chapter 2 will discuss the background of fingerprinting and watermarking, a related IP protection technique, as well as other IC

security primitives that are similar to fingerprinting. Chapters 3-6 of the dissertation will describe our new techniques for giving a circuit an individualistic fingerprint. The first method attempts to create fingerprints by using observability don't cares, which occur in almost every combinational circuit to create a fingerprint. The second method utilizes satisfiability don't cares which also occur frequently in ICs to generate unique fingerprints. The third method will use the unused states and transitions in an incompletely specified finite state machine to create a fingerprint. Our final method uses scan chain interconnects to create individual fingerprints. Chapter 7 will explain some future work and finally in Chapter 8, the dissertation conclusions will be presented.

CHAPTER 2: SURVEY OF CURRENT WORK

Currently in the field of IP protection, there is more new work being done in watermarking than in fingerprinting. Fingerprinting, while a fairly old concept, has not had as much literature written about it, especially for non-FPGA designs. First we will go over the details of watermarking and fingerprinting, their similarities and differences, then I will discuss briefly the state of the art of watermarking, and finally I will go into the current work being done in the field of fingerprinting for both ASICs and FPGAs.

Both watermarking and fingerprinting are considered to be "Indirect Protection Techniques" as defined by [10]. These techniques have a list of properties that they should exhibit:

i. *Proof of Authorship and Authenticity*: the signature should unambiguously identify the owner and be detectable
ii. *Correctness of functionality*: the added functionality should not affect the correctness of the functionality
iii. *Low overhead*: The required hardware overhead should be low
iv. *Ease of detection*: The signature should be detectable with a low cost technique

For these properties, there is a list of metrics that are also defined that help evaluate different protection techniques

i. P_u, probability of uniqueness: the odds that another design carries an identical signature.

ii. P_m, probability of a miss: the odds that the signature is not detected (although exists) due to the tampering.

iii. P_f, probability of false alarm: the odds that a signature is detected (although exists) unintentionally. In general $P_f = P_u$

These two lists help us to determine what would be a good watermarking or fingerprinting technique, by giving us objectives to strive for. The first list is our main objective and we modify our techniques to improve the metrics P_u, P_f, and P_m. The work in [10] also defines what watermarking and fingerprinting are:

Definition 1. *A fingerprint is defined as a signature generated from a given design without altering it in such a way that identifies it uniquely.*

Definition 2. *A watermark is defined as a signature embedded in a design using a non-invariant mapping in such a way that it identifies the owner uniquely*.

From this, one can think of a fingerprint as a change in a circuit that causes no functional difference, but can still be observed, whereas a watermark will make changes to the output, but not to the outputs that determine functionality. Fingerprints can also be used to increase P_u by adding more changes, and using specific changes as variables during manufacturing.

A. IP Protection in Digital Media

Before IP protection was a major concern for ICs, it was researched thoroughly for use in digital media. In 1993, the term "Digital Watermark" was coined in a paper by Schyndel et. al [11]. Similar to IC protection, the goal of digital watermarking is to

"provide and ensure security, data authentication, and copyright protection to the digital media" [12]. This is done by inserting secret information, identifiable by the media owner, into the digital data. There are several types of basic robust watermarks that are difficult to remove using common media operations, including noise watermarks, logo watermarks, and message watermarks. These watermarks are usually seen by the user to alert them of the owner/creator of the data. There are numerous other types of watermarks that use more complicated techniques that edit bit level data either deterministically or randomly.

In addition to watermarking, fingerprinting has also been explored in the digital media domain. An original taxonomy for fingerprints is given in [13] where the author also suggests adding fingerprints to computer software. Later, work started to emerge in marking digital video and images [14], [15], [16], [17], [18], documents [19], and computer programs [20]. In [19], it is shown that these marks can be combined to create fingerprints that are collusion resistant, such that if two adversaries were to compare their data, it would be difficult to determine where the fingerprint exists.

B. Watermarking ICs

The first main method of IP protection, for ICs, is that of embedding a watermark into a circuit. This is a similar concept to that of putting a watermark in digital or physical media. The difficulty of this comes from trying to obscure the actual location of the watermark from an adversary. For digital media, this is fairly different task because it is difficult for a human to see where watermarks have been injected, especially if they are hidden in the data.

One of the older methods of watermarking for integrated circuits (ICs), developed by Charbon in [21], requires that watermarks are put in a circuit in different steps of the synthesis and layout processes. This gives many options for the kinds of watermarks that are put in and in what layer of the design. In addition, every layer after the one the watermark is placed in is then protected by the watermark. This is made easier by the work in [4], which gives techniques for forcing specific characteristics onto the circuit from the design process.

The work in [22] changes this, by introducing a method for injecting a watermark in the finite state machine (FSM) or state transition diagram (STG). The author changes the way the state transitions behave as well as the unspecified transitions such. The goal of this is to make the circuit behavior individualistic such that if someone else developed the same circuit, they would not stumble upon the same watermark. Then the work in [23] takes this further by using the unspecified states and transitions to create a specific input/output pattern so that a signature may be programmed into the circuit during its design. This method included adding new states or inputs to increase the number of transitions that can be manipulated.

In order to use these watermarks in a court room setting, designers will use a third party which generates a watermark and keeps a record of it. The designer creates an IP, submits the authorship information to the third party and gets a signature to put into their IP, using one of the techniques mentioned above. When an adversary copies their circuit, the author can then extract the signature and prove that the circuit is theirs.

Although watermarks are good for proving ownership of a device with a great probability, they have a significant problem. To prove in court that a device has the watermark of a designer, it must be publically shown. Once this watermark is shown, it can be rendered useless by a clever adversary. This would require the designer to recreate the original circuit with a new watermark which would be expensive.

C. Fingerprinting IP

Circuit fingerprinting attempts to fix the issue of single use watermarks by using individual characteristics of a circuit. Like a human fingerprints, which can identify any one person, circuit fingerprints attempt to identify individual ICs, or batches of ICs, that were manufactured. These can either be placed in the circuit beforehand or derived from process variations and other circuit characteristics.

1. Alternative Fingerprinting Uses

Patel et al. created a technique in [24] to determine a circuit's fingerprint through its glitches. A glitch may be considered a temporary signal that is incorrect but overall does not affect the performance of the device. The authors state that if the glitches are not causing major issues in the circuit, that they can be used as a fingerprint. In addition, by changing the operation temperature they are able to create more temporary glitches that could help them identify specific ICs.

Agrawal et al. in [25] and Jin et al. in [26], use different techniques that takes advantage of process variations as opposed to glitches that can be fixed. They use the side channel information, power and delay path variations, respectively, to create a

fingerprint for a circuit. This helps them prevent Hardware Trojans, which would increase the path delay and change the output of the IC in a harmful way.

The emerging circuit function, the physical unclonable function (PUF), must also be discussed. Based on the unclonable intrinsic fabrication variation in delay, capacitance, or threshold voltage, the PUF circuitry can produce a unique response to a given challenge to authenticate a device or generate a bitstream as the cryptographic key, and is similar in architecture to the work in [24] and [26]. PUF is some additional circuitry and thus it will not alter the functionality. Furthermore, the fabrication variation is believed to be unique. So the first two requirements of fingerprints we proposed earlier are satisfied. However, when an IP is illegally reproduced, the PUF information will be changed and thus violate the last requirement for fingerprint. Therefore, PUF can be used as a fingerprint for device authentication, but not for IP protection.

The work in [24] and [26], and the work being done in PUFs are techniques designed for individual circuits. None are techniques that are suitable for IP protection because if an adversary is willing to copy a circuit, the fingerprint will be completely different from the fingerprint in the circuit that they copied. If this is the case, someone trying to prevent duplication would not be able to prove that an adversary is copying their work because they would not have a record of the new fingerprint coming from their designs.

Figure 4 is an illustration of this concept. We see at the top level the "golden copy" of our circuit that strictly contains the functionality that we design. At the next level, the legitimate copies, are the copies that contain a fingerprint or PUF and are

distributed to the various customers. At this level, each circuit has an individual identifier, in this case a number between 1 and n. The next level is where the adversary tries to copy the circuit with the identifier 1. For the PUF example each copy now has a new identifier as designated by the 1' to n'. In the case of the fingerprint though, each copy has the same exact identifier as the one before it, leading back to the original legitimate copy. This is the reason that it is possible to track the counterfeit or re-packaged devices back to a registered customer with a fingerprint and not a PUF or fingerprint as used in [24] and [26].

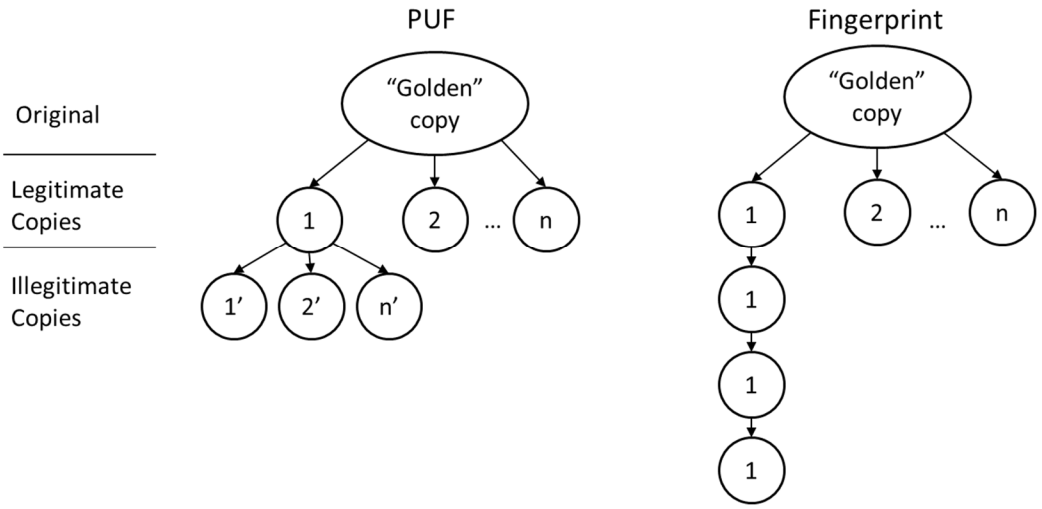


Figure 4. The result of copying a circuit with a PUF vs. a Fingerprint

2. Fingerprinting for FPGA IP Theft Protection

Closer to the topic of this dissertation, is the concept of fingerprinting for Field Programmable Gate Arrays (FPGAs). This work was started in [27] and [28], and in this paper the authors propose a concept similar to the ones presented in this dissertation, except for FPGAs. Their concept was to take advantage of the FPGA architecture to create small changes in the implementation of an FPGA core to create a watermark. FPGAs are made up of thousands or millions of look up tables which

are programmed to behave in a specific manner. The authors of this work rearranged where certain functions were placed in various locations to create a fingerprint.

Figure 5 provides a small example of one such change. Each of the four blocks represent the same function Y = (AB) + (CD), and use the same amount of resources. In each block a single a single configurable logic block (CLB) is free and each CLB contains a LUT where a signature can be placed. The blocks where this signature are varied so that adversaries have difficulty colluding to defeat the fingerprint.

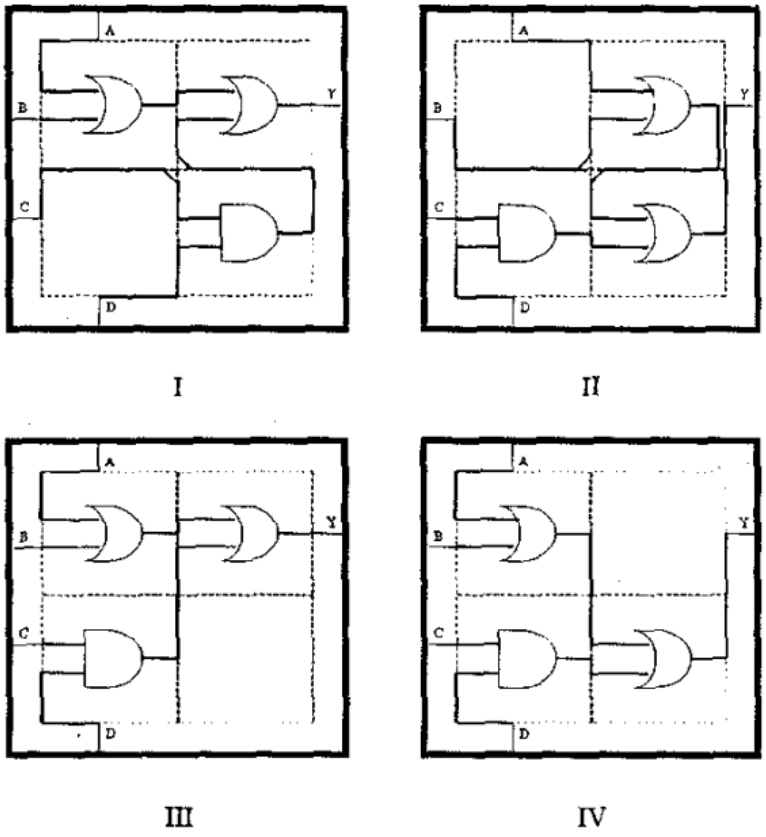


Figure 5. Example of small FPGA change utilizing LUTs [27]

Other ideas for fingerprinting or watermarking FPGA cores have included encrypting the bit stream file [29] and adjusting the timing constraints to adjust a small part of the design, for example in the post and route phase [30].

Fingerprinting for FPGAs is a significantly different process than for ASICs. The cores are simply stored as bitstream data, and as such it is easy to manipulate them or make several copies with unique fingerprints. Fingerprints can be added in at various stages of the core design, especially if certain aspects of the design are forced and not optimized by whatever software is being used to generate the bitstream.

3. Fingerprinting for ASIC IP Theft Protection

Due to its difficulty, fingerprinting of ASICs for IP protection, does not have much research being done as watermarking. The first major paper on this specific topic was done by Caldwell et al. of [3], where the authors attempt to create fingerprints by using "specific VLSI CAD optimization" heuristics. They use the NP-hard problems of partitioning, satisfiability, graph coloring and standard-cell placement problems, similar to the watermarking work in [31], [2], [32], and [12], with different criteria to create independent fingerprints for each buyer of the device.

Building off the idea of the idea in [3], a conceptually different fingerprinting method for the graph coloring problem is proposed in [33], where the authors effectively add new constraints to the graph by either adding new states or adding new pathways which either manipulate cliques or bridge current nodes. By doing this, they are able to increase the solution space which means that they can create even more varied fingerprints, which was a possible problem of [3] if the design was highly specified before optimization.

The issue with both [3] and [33] is that these methods must be implemented in the earlier stages of the VLSI design cycle. This requires a significant amount of redesign to the circuit and will increase the production cost and time. This is where this dissertation makes its contribution and improvements to the state-of-the-art.

CHAPTER 3: OBSERVABILITY DON'T CARE FINGERPRINTING

Our first work in this dissertation was to develop a method to create fingerprints in circuits using Observability Don't Care conditions (ODCs) in circuits. Observability don't cares (ODCs) can be found in almost any circuit that has more than a trivial number of logic gates, and thus is an excellent method for creating a circuit fingerprint. In addition, our method generate little overhead and a large number of potential fingerprints for ICs of a significant size.

A. Introduction to ODC fingerprinting

To start we first use a small example to show the basic idea of ODC fingerprinting.

Illustrative example.

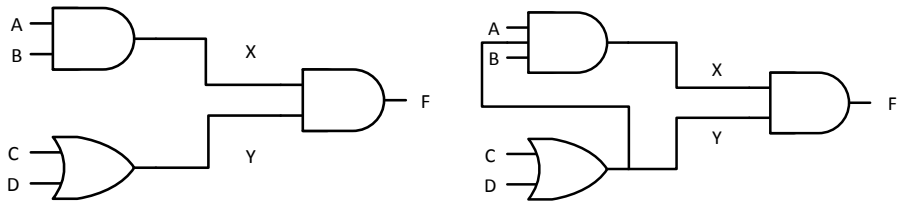


Figure 6. Two 4-input circuits that implement the same function.

The left circuit in Figure 6 realizes the function $(AB)(C + D) = F$. When the Y input to the AND is zero, the output F will be zero regardless of the value of X input; however, when Y=1, F will be determined by the X input. So when we direct signal Y to the AND gate that generates X, as shown in the right of Figure 6, we can easily verify that this circuit implements the same function F. However, these two circuits are clearly distinct. Moreover, if one makes a copy of any of these circuits, this distinction remains. Thus we can embed one bit fingerprint information by controlling

whether X depends on Y. This fingerprint meets all the three requirements we listed earlier.

One key feature of this approach is that the changes we make on the circuit are minute. We can make a connection, as shown on the right circuit in Figure 6, during routing and placement; then determine whether to keep this connection based on the fingerprint bits at post-silicon phase. This avoids the expensive redesign and fabrication based on a new layouts as in the current fingerprinting approaches [3] [33].

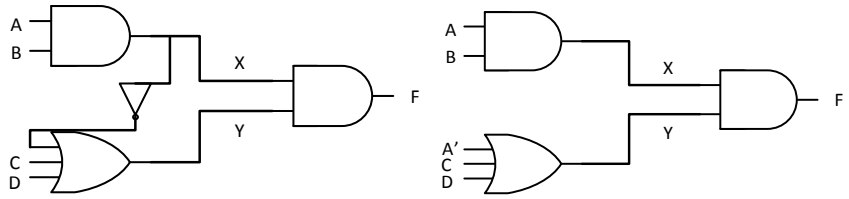


Figure 7. Two more implementation of the same function.

Challenges and contributions. As simple as the above example suggests, there remain several technically challenging questions to develop a systematic fingerprinting technique based on this idea. For example, the two circuits shown in Figure 7 also implement the same function F as those circuits in Figure 6 and they all have similar layout. The contribution of this paper is the discovery of the proposed practical fingerprinting method and its implementation.

First, how to efficiently identify the locations in the circuit where we can make changes to reflect the fingerprint? Second, at such fingerprinting locations, what kind of changes to the circuit can we make? Third, how much fingerprinting information can be embedded, or how many different fingerprinting copies can be generated, by

this approach? Finally, what is the impact to the design quality such as area, delay, and power after embedding fingerprints?

We propose to compute the observability don't care (ODC) conditions as each gate and use such information to locate the gates that can be modified to embed fingerprints. Once the fingerprinting locations are identified, we analyze the circuit locally to determine what kinds of changes can be made. Because ODC conditions exist almost everywhere in any combinational circuit, this provides us a large space to embed fingerprints. When we design circuits by the consideration of adding fingerprints after fabrication, we realize that reasonable area and power overhead have incurred, but the delay penalty is too large to accept. Therefore, we propose a delay-driven heuristics to create fingerprinting copies under delay constraints.

B. Observability Don't Cares

In this work we use the concept of observability don't cares (ODCs) to create fingerprints, so first we will go over the concept of an ODC.

Observability don't cares are a concept in Boolean computation. The conditions by which an ODC occurs are when local signal changes cannot be observed at a primary output. An example of this can be seen in Figure 8 below. When the bottom AND gate has an input equal to zero, it will generate a zero as its output. This output will be propagated to the next AND gate and generate another zero as the primary output for this circuit. The signals from C, A, and B are all blocked and cannot be observed from the primary output.

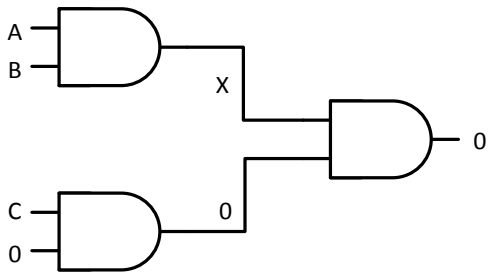


Figure 8. ODC on the AND gate

ODCs can be several layers deep and can cause several different signals to be blocked, depending on the input to the circuit.

Formally, the ODC conditions of a function *F* with respect to one of its input signal *x* can be defined as the following Boolean difference:

$$ODC_x = \left(\frac{\partial F}{\partial x}\right)' = \left(F_x \oplus F_{x'}\right)' = F_x F_{x'} + F_x{}' F_{x'}{}' \quad (1)$$

This equation has a few parts to define. First we have ODC_x which simply represents if we have an ODC that occurs at signal *x*, essentially that is there some condition that causes signal *x* to be ignored. The next part of the equation $\left(\frac{\partial F}{\partial x}\right)'$ states that what we are looking for when determining ODC_x is when the output of function F does not change despite a change in the signal *x*. The next two pieces of this equation, $\left(F_x \oplus F_{x'}\right)'$ and $F_x F_{x'} + F_x{}' F_{x'}{}'$ represent the actual functional situation where and ODC occurs at *x*. F_x represents the normal output of function *F* when *x* = 1 and $F_{x'}$ represents the output of *F* when x = 0. The next two symbols $F_x{}'$ and $F_{x'}{}'$ simply represent the inverted output of F_x and $F_{x'}$.

Basically (1) states that when we have a function *F*, and a variable *x*, when the condition ODC_x is satisfied, the value of variable *x* will not have any impact to the

value of the function *F*. In the above example, the final output is produced by a 2-input AND gate, for one of its input *x*, if the other input is *y*, then from equation (1), we have $ODC_x = y'$. Hence, when the other input has value zero (as shown in Figure 8), input *x* becomes an ODC.

Using this equation we are able to derive a list of ODC calculations for the gates in the library we use in the experiments for this work (packaged with SIS [34]), as seen in Table 2. ODC calculations for the library cells.. Gates such as inverters, XORs, and XNORs do not have ODCs and are omitted from the table. Other gates such as 3- and 4-input NORs and NANDs, and 4-6 input AOIs and OAIs are simple extensions of the gates listed in Table 2. For NAND, NOR, AOI22, and OAI22, all the inputs are symmetric, so we only compute the ODC for one of the input. For gates like AOI21 and OAI21 that have asymmetric inputs, we compute ODC for all the asymmetric inputs.

Table 2. ODC calculations for the library cells.

Gate	***Gate Equation***	***ODC Calculation***
NAND	$O = \overline{AB}$	$ODC_A = \bar{B}$
NOR	$O = \overline{A + B}$	$ODC_A = B$
AOI22	$O = AB + CD$	$ODC_A = \bar{B} + CD$
OAI22	$O = (A + B)(C + D)$	$ODC_A = B + \bar{C}\bar{D}$
AOI21	$O = A + BC$	$ODC_A = BC$
		$ODC_B = A$
OAI21	$O = A(B + C)$	$ODC_A = \bar{B}\bar{C}$
		$ODC_B = \bar{A}$

C. ODC Fingerprinting

The goal of this work is to find a way to improve upon current fingerprinting techniques. Most current techniques, such as the work in [24] and [26], use process variation as a method of fingerprinting a circuit. This can be unreliable because it requires significantly accurate measurements to be made, both when determining the

fingerprint initially and any time afterward. These are passive techniques so if an adversary copies the circuit, the owner's fingerprint is no longer present, which violates the *heredity* requirement for fingerprinting and defeats the purpose of the fingerprint. Other methods like those mentioned in [3] and [33] are significantly more costly due to the need for high-level circuit redesign.

Our method attempts to use ODCs to create small changes in the circuit which can be implemented in the later stages of the VLSI design cycle. These changes will have no effect on the functionality of the circuit, but will allow a designer or manufacturer to determine the circuit's origin. They also allow the designer to create specific fingerprints for groups of ICs or individual ICs, depending on the IC's design and buyer information.

Although this work is centered on the concept of utilizing ODCs to create small modifications throughout a circuit, several challenges prevent it from being a simple task. First, not all gates that create an ODC can be used to create a circuit modification, so conditions need to be established to find locations suitable for modification. Once a location is established, a modification may be made. The second challenge is that the modification is dependent on the location and gates at that location. Finally, once all of the modifications have been made, it is important to prevent unacceptable overhead from occurring due to the modifications. In the rest of this section, we elaborate formally these challenges and our solutions to solve them.

1. Finding Locations for Circuit Modification based on ODCs

Every logic circuit that is created uses a library of gates that determines the logical relationships that can occur. Most libraries contain gates that create ODCs as defined

using Equation (1), but not every instance of these gates will be able to be modified to accommodate a fingerprint. There are four necessary conditions that must be met for a gate to be considered a fingerprint location and are enumerated in the following definition.

Definition 3. (Fingerprint location). *A fingerprint location is defined as two or more gates that can be considered for modification for a circuit fingerprint without changing the functionality of the circuit. These gates consist of a single primary gate and one or more gates that generate inputs for the primary gate that meet the following criteria:*

1. *The primary gate must have at least one input that is not a primary input of the circuit.*
2. *The primary gate must have at least one input which is the output signal of a fanout free cone (FFC), which means that this signal only goes into the primary gate.*
3. *The FFC in criterion 2 must have either a gate with non-zero ODC (such as those in Table 2) or a single input gate (e.g. an inverter).*
4. *The primary gate must have a non-zero ODC with respect to one or more of its input signals other than the one from the FFC.*

Criterion 1 is necessary for making local minor changes to the circuit (for fingerprinting purpose). Criterion 2 ensures that the changes made to the FFC will not affect the functionality of the circuit elsewhere. Criteria 3 and 4 provide a possible signal (in criterion 4) that can be added to a gate in the FFC (in criterion 3). Each ODC gate, in a circuit is analyzed using Definition 3 and if it satisfies all the criteria

in the definition, it is then considered to be a fingerprint location, a location where the circuit can be modified to add the fingerprint.

2. Determining Potential Fingerprint Modifications

For each fingerprint location that is found, a modification can be applied to the gate's inputs. A generic modification for a fingerprint is depicted in Figure 9.

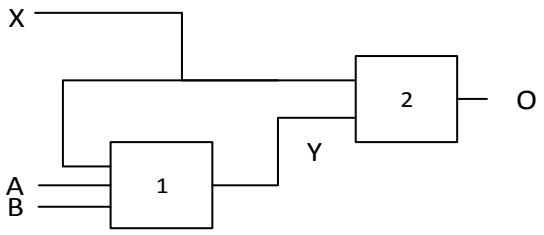


Figure 9. Generic fingerprint change

Figure 9 has two generic gates, represented as boxes 1 and 2, three primary inputs (X, A, and B), and one primary output (O). Gate 2 represents the primary gate, gate 1 represents the gate within the FFC that generates signal Y, and signal X is independent of the FFC that generates signal Y. Suppose that signal X satisfies ODC_Y, thus we can add signal X into the FFC of Y, for example gate 1 as shown in Figure 9, either in its regular form X or its complement form X'. However, when we make this addition, we need to guarantee that when signal X takes the value that does not satisfy ODC_Y, it will not change the correct output value Y. In the rest of this paper, signal X will be known as an ODC trigger signal, as defined below

Definition 4. **(ODC Trigger Signal)** *An ODC Trigger signal is a signal that feeds into a gate, with a non-zero set of ODC conditions, which causes the ODC condition to activate. In the context of this work it also*

represents the signal that is used to modify the input gate to the primary gate for the fingerprint modification.

In order for this to work, the relationship between the signal X, gate 2, and gate 1 must be analyzed so that X only changes gate 1's output, Y, when it also triggers the ODC, criterion 3 in the definition of a fingerprint location. For every possible pair of gates that can be considered a fingerprint location, similar to gate 1 and gate 2, a structural change must be proposed in order to modify that location. This requires a maximum of n^2 proposed changes, where n is the number of ODC and single input gates in a library. An example of this exists for the library we used, in Table 3, later in this chapter. Modifications with certain gates may not always be feasible, especially if the overhead costs are too large.

For simple changes like the one in Figure 9 or those in the motivation example, each location like this can be considered a position to embed one bit in a bit string that represents the fingerprint. For each circuit that is manufactured, this fingerprint location can be either modified 1, or left alone 0. This means that for a circuit for n potential fingerprint locations there are at least 2^n possible fingerprints and n bits of data in the bit string.

In addition to the simple change in Figure 9, a fingerprint location may also consist of more than one input gate as stated in Definition 3. If this is the case, a modification similar to the one in Figure 9 may be applied to each of the input gate in the FFC that abides by Definition 3, criterion 3. If this situation occurs, k bits are added to the fingerprint bit string, and the number of potential fingerprints is multiplied by 2^k, where k is equal to the number of input gates in the FFC of the primary gate.

It is also possible to leverage certain gate combinations to add data to our fingerprint bit string and also to reduce the delay that can be caused by rerouting signals. If the ODC gate that is being considered as a fingerprint location, is immediately preceded by another ODC gate with the same ODC trigger signal, such as the two ANDs in Figure 10 or for example a NOR preceded by an AND, the fact that the ODC would be triggered before the X signal is generated in Figure 10 can be utilized to directly feed one or two input signals into the gate from the fan out free cone. The OR gate in Figure 10 shows an example of this. Input signals, A or B are simply inverted and directed into the OR gate, removing the need to put X into the OR gate which cause a delay if the gates on the left hand side have equal delay.

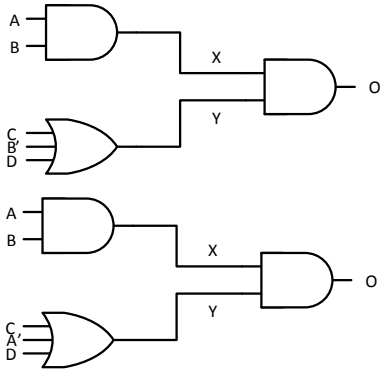


Figure 10. Fingerprint change that reroutes earlier signals

This additional method of fingerprint change allows us to add more data to our fingerprint bit string. For every gate combination like this we are able to add at least $n(n+1)/2$ potential changes, where n is the number of inputs to the ODC trigger gate, because all combinations of the inputs to the ODC trigger gate can be added to the inputs of the fan out free cone gate. This also means we can add $\log_2(n(n+1)/2)$ bits to our fingerprint bit string. In sum, with only one fingerprint location, we

may be able to modify the circuit in potentially many different ways and thus embed multiple bits of fingerprint.

3. Maintaining Overhead Constraints

The fingerprint modifications proposed can cause a large overhead, relative to the circuit's initial performance. Rerouting paths, increasing the input size to input gates, and introducing new inverters are the cause of the overhead. Two heuristic methods have been considered for reducing this overhead, a reactive method and proactive method.

Of the two methods, the reactive method is easier to implement but is difficult to scale. This method involves taking a fully fingerprinted circuit and by removing one fingerprint modification at a time, analyzing the difference in overhead, whether it be area, delay, power, or something else. The modification that results in the largest change to the overhead is removed and the resulting circuit is tested again. This is done until a certain overhead constraint is met or there are no more modifications to remove.

The proactive method is more difficult to implement, but because it is done as modifications are applied it scales well with larger circuits. This heuristic requires that each modification is analyzed before being implemented. For area and power this is simple because any new gates or changes in gates will result in overhead that can be estimated using information about the cells in the library. Delay is more difficult to analyze because not every modification will slow the circuit down. As modifications are added the critical path may change which changes where new modifications should be considered. The delay can be estimated by determining the

slack on each gate and updating the information every time a modification is made, but this can be time consuming for large gates that will have a large number of modifications. For this proactive method, modifications would be added until a certain overhead constraint was met, the opposite of the reactive method.

4. Security Analysis

As we have mentioned in the introduction, an IP will be protected by both watermark (to establish the IP's authorship) and fingerprint (to identify each IP buyer). When a suspicious IP is found, the watermark will be first verified to confirm that an IP piracy indeed happens. Next, the fingerprint needs to be discovered to trace the IP buyer who may be involved in the IP piracy. It is trivial for the IP designer to detect the fingerprint embedded by our proposed approach because the designer can compare the fingerprinted IP with the design that does not have any fingerprint to check whether and what change has occurred in each fingerprint location to obtain the fingerprint.

However, it is infeasible for an attacker to reveal the fingerprint locations from a single copy of the IC. This is because that when the fingerprint information is embedded at a fingerprint location, the FFC of the fingerprinted IP will include the signal that is not in the FFC in the original design when the fingerprint location is identified. Consider the left circuit in Figure 6 of the motivational example, the FFC that generates signal X contains only the 2-input AND gate with A and B as input. But when signal Y is added to this AND gate, the FFC will include the 2-input OR gate with C and D as input. This will make this portion of the circuit not a fingerprint location (criterion 4 is violated).

When the attacker has multiple copies of fingerprinted ICs, he can compare the layout of these ICs and identify the fingerprint locations where different fingerprint bits were embedded in these ICs. This collusion attack is a powerful attack for all known fingerprinting methods. Careful designed fingerprinted copies distribution scheme may help [3] [33] [31], but requires a large amount of fingerprinting copies. As we will demonstrate through experiments, our proposed approach has this capability and thus can reduce the damage of collusion attack. In addition, it is also known that as long as the collusion attacker does not remove all the fingerprint information, all the copies that are involved in the collusion can be traced [3] [33] [31].

D. Experimental Setup

To prove the usefulness of this new method, a circuit modifier was constructed in C++ based on the description in the previous section. We started with the Microelectronics Center of North Carolina (MCNC) benchmark circuits in the Berkeley Logical Interchange Format (blif), which specifies the circuits' logical behavior, not its physical layout. From here they were put through Berkeley's ABC [35] program with a library of gate cells. The ABC program can map a blif file to a Verilog netlist with the standard gates in the library. ABC also allowed us to get both the area and delay of the benchmark circuit.

1. Initial Fingerprinting

Our first goal was to create a program that could implement the details of section IV A and B. We started by gathering all of the gate data, from the benchmarks, including: gate type, fan outs, fan ins, and gate depth like in Figure 11.

Combining this information with the ODC calculations in Table 2 allowed us to determine what gate combinations were viable fingerprint locations.

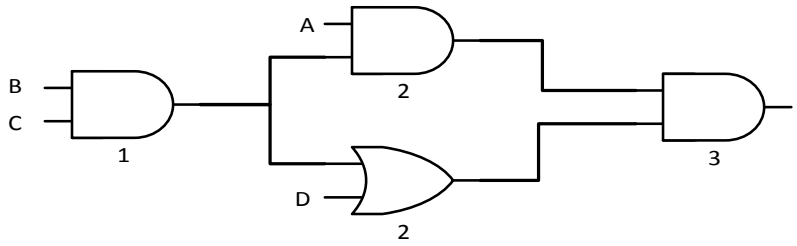


Figure 11. Gate depth examples

```
Input: Circuit in Verilog netlist format
Output: Circuit in Verilog netlist format with fingerprints inserted
Variables: Gi stores gate information for each gate
1)  for each line in file:
2)      if(line == gate)
3)          add gate to Gi
4)  for each gate in Gi:
5)      store gate type, fan ins, fan outs in Gi
6)      determine and store depth in Gi
7)      if(gate creates ODC):
8)          for each gate2 that fans into gate
9)              if gate2 only feeds into gate
10)             mark gate2
11) for each gate in Gi:
12)     if(gate has marked fan ins && creates an ODC)
13)         choose fan in with greatest depth
14)         choose other gate with lowest depth
15)         add fingerprint to new file
16)         continue
17)     else
18)          Write original gate to new file
19) output new file
```

Figure 12. Pseudo-code of proposed method

The next problem was to take the fingerprint locations and modify them to get the maximum fingerprint size. For each fingerprint location we chose to work with the input gate, within the fan out free cone, which had the highest depth. We chose the gate with the highest depth to our primary gate so that we knew it did not need a signal until the latest possible time, reducing delay. For the input signal, we chose the ODC trigger signal that occurred at the earliest depth to create our fingerprint

location. The ODC trigger signal was chosen so that we could reduce our delay overhead. Once all of the fingerprint locations were discovered, a look up table was used to determine which fingerprint modifications could be made to the circuit. Figure 12 shows this entire process in pseudo-code and Table 3 shows the potential fingerprint modifications.

Table 3. List of possible changes to a gate that feeds into a NAND ODC, based on the library used

Feed In Gates	*New Feed In Gates*
inv	nand2
nor2	nor3 with inv
nor3	nor4 with inv
nand2	nand3
nand3	nand4
aoi21	aoi22
aoi31	aoi32
aoi32	aoi33
aoi22	aoi221
aoi211	aoi221
aoi221	aoi222
oai21	oai22 with inv
oai31	oai32 with inv
oai32	oai33 with inv
oai22	oai221 with inv
oai211	oai221 with inv
oai221	oai222 with inv

Each of benchmark circuit netlists were run through our circuit modifier and the resulting performance metrics (area, delay, and power) were recorded. Table 4 is a listing of a subset of the benchmark circuits that were tested as well as their original area, delay, and power measurements, in columns 3-5. After the fingerprint modifications were applied, we re-measured the circuit, and got the new area, delay, and power, as seen in columns 8-10.

E. Reduction of Overhead

Our original results, shown in Table 4, columns 6-10, had a large delay overhead. To compensate for this overhead, a program was written to implement the reactive method, mentioned in section C.3 of this chapter, tuned for delay overhead. This method was chosen because it gave us an approximate upper bound on fingerprint locations and did not require any sophisticated industrial tools.

The program went through our design and took turns removing each fingerprint location we created, one at a time, and tested the result against the original circuit. The result with the minimum delay was saved as our new circuit, and the process started over again with the new design as the base design. The program would then continue to attempt to remove finger print locations, one at a time, until we reached our overhead constraint.

When no fingerprint location could be found that reduced the system's delay, random fingerprint locations were removed until a better delay could be achieved again. Because this was done at random, not systematically, this program needed to be run several times to find more optimal solutions. As a result, the data we obtained may not be the optimal solution, but the data usually left a good number of fingerprint locations for each gate of a significant size.

F. Experimental Results

As we can see from the results, our area overhead is acceptable except in a few of the circuits. The largest area overhead is for one of the smallest circuits and it can be reasoned that even small changes to the gates and possibly including new ones will

cause a significant change in the area. The bottom of Table 4 shows our average results and an average of 12.60% overhead is acceptable, especially when a number of circuits have between 20 and 400 fingerprint locations that were changed.

On the downside though, the delay overhead is fairly poor. Table 4 shows that we have an average delay of 64.36%. At its worst it is almost 80%, which means the circuit would need to run at almost half its original speed. This is unacceptable. The reason for this result is that the circuit modifying program does not yet check for changes that are made on the critical path, where changes would cause increases to the delay.

The major result of this work, is the number of possible fingerprint locations. Both the circuit size and gate composition contributed greatly to the number of fingerprint locations. In circuits with a smaller number of gates, more variation occurred in the number of fingerprint locations. This is because the layouts are more varied as well. A circuit with a shallower depth will not have as many gates that have inputs from previous gates.

For the rest of the gates, the number of fingerprint locations was quite good. The number of locations allows for a minimum of 2^n possible fingerprint combinations, where n is the number of locations. This means that even if we only have 21 locations, as in gate C432, we can create more than 2 million possible fingerprints.

The reason we say that 2^n is a minimum number is because for every fingerprint location, there can be several configurations, as discussed in section C. These configurations can increase the total number of fingerprints significantly, as can be

seen in the column six of Table 4. Column 7 of Table 4 shows the $log_2(n)$ where n is the number of possible fingerprint combinations there are. This was done for two reasons: 1. the numbers were so large in some cases that the data could not be accurately represented in our tables and in the program we wrote and 2. this data gives an idea of how much information can be hidden in these fingerprints if you think of each combination as a bit of information. For most of the circuits, the number of possible combinations of changes to the circuit is far larger than 2^n.

For gates with an excessive number of fingerprint combinations, we can either eliminate some of the locations to reduce our overhead, or include additional functionality to our fingerprints, such as error correcting codes or redundancy, so that even if an adversary tampers with the circuit, we can figure out what they have done and what the original fingerprint was.

Table 5 shows the average results of our heuristic approach to delay overhead management. The rows show we put on a 10%, 5%, and 1% delay overhead constraint on delay. For most of the larger circuits we still had a good number of fingerprint locations left to create a robust fingerprint set. Our area, delay, and power overhead are also minimized. These gates had few fingerprint locations to begin with so attempting to put a delay constraint on them was not possible with the current system. Figure 13 also shows a comparison in terms of fingerprint size for our original results versus the results of our delay constrained fingerprint. It can be seen that, while there is a steep decline in fingerprint size when constraining the delay, the size of the fingerprint for the 5% and 10% delay constraint are still of a significant

size. In addition, for the larger circuits, the 1% constraint still provides us with a large fingerprint.

Table 4. Results of a subset of MCNC/ISCAS 85 and 89 benchmarks before/after ODC fingerprint injection.

	Gate Count (original)	*Area*	*Delay*	*Power*	*Fingerprint Locations*	*Log_2 (Possible Fingerprint Combinations)*	*Area Overhead*	*Delay Overhead*	*Power Overhead*
C432	166	269584	9.49	1349.5	40	68.07	11.19%	54.69%	6.05%
C499	409	662128	7.62	2951.6	112	177.16	9.25%	31.23%	10.00%
C880	255	426880	6.95	2068	38	66.58	6.52%	47.05%	5.86%
C1355	412	668160	7.67	2988.2	118	187.36	9.86%	30.38%	9.44%
C1908	395	635216	10.66	2655.4	88	151.25	11.40%	46.53%	11.92%
C3540	851	1469488	11.64	7242.3	179	376.79	10.10%	50.52%	9.46%
C6288	3056	4797760	32.92	1	420	635.26	6.29%	34.33%	N/A
des	3544	5831552	6.64	23145.3	782	1438.62	11.87%	75.00%	8.13%
k2	1206	2039280	5.82	5482.4	241	470.25	13.36%	78.87%	8.64%
t481	826	1478768	6.49	4188.1	178	418.62	13.49%	74.42%	7.08%
i10	1600	2676816	12.65	9729.9	316	601.15	9.85%	48.70%	9.03%
i8	1211	2273600	4.73	9621.6	235	541.13	9.45%	67.44%	10.63%
dalu	836	1383184	10.1	5275	298	507.57	15.97%	47.13%	21.45%
vda	635	1088080	4.51	3270.4	134	277.42	14.24%	58.98%	9.75%
Avg Change							12.60%	64.36%	10.67%

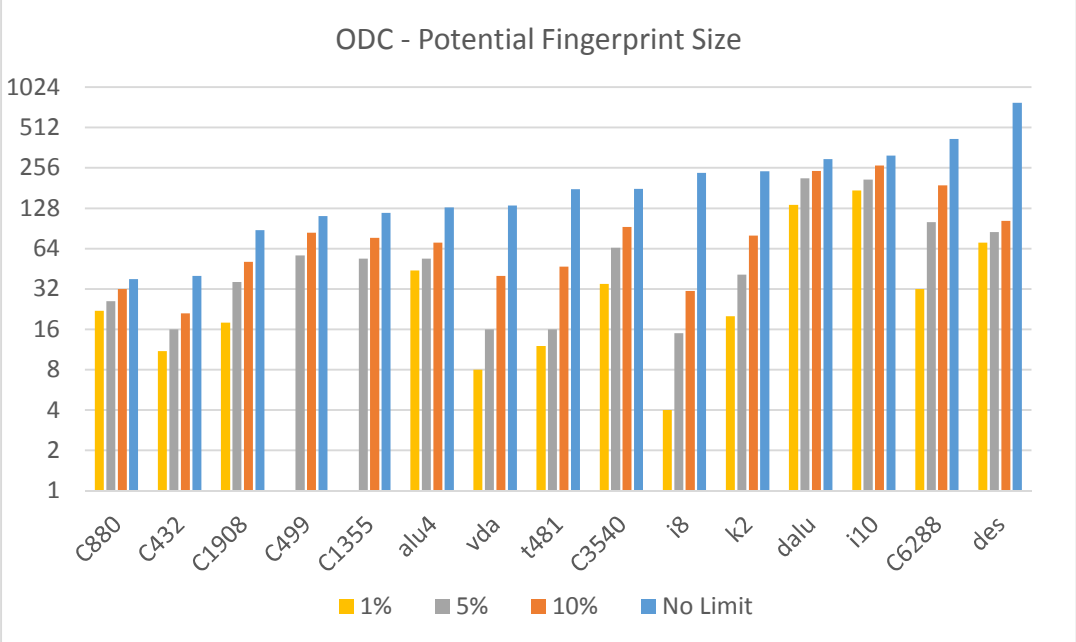


Figure 13. Fingerprint sizes for tested circuits before and after constraints

Table 5. Average results after heuristic overhead constraint

	Fingerprint Reduction	*Area Overhead*	*Delay Overhead*	*Power Overhead*
10% Delay Constraint	49.00%	*5.04%*	***9.42%***	*4.99%*
5% Delay Constraint	64.30%	*3.57%*	***4.44%***	*2.46%*
1% Delay Constraint	81.03%	*2.40%*	***0.41%***	*2.65%*

G. Conclusion

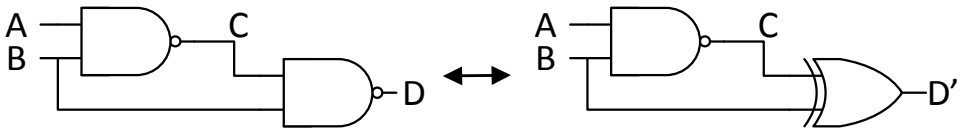
This work has shown that we can create a significant number of fingerprints that are hard to detect, and thus hard to remove for circuits of a significant size. These fingerprints have a minimal overhead since they only require a minor change in the gate level design. In addition, if we utilize the amount of data we can hold in these fingerprints, we will have the opportunity to make the fingerprints more robust against attackers. There is potential future work, where we can look at how we can implement these fingerprints further down the manufacturing line. Potential methods include using fuses as the connections for the added lines so we can decide which ones are active, determining where there are empty lanes to where we can add fingerprint locations and be able to add connections after fabricating the rest of the circuit or using engineering changes to remove or reroute lines in post-production.

CHAPTER 4: SATISFIABILITY DON'T CARE FINGERPRINTING

An alternative to using ODCs to create fingerprints is using the Satisfiability Don't Cares (SDCs) that occur in circuits. These too can be used to create small changes in a circuit so that you can create a fingerprint that causes no functional change to a circuit, creates a minimal overhead, and allows a designer to identify where their work is being used. There is also a potential to create holes in a design where depending on the fingerprint, different gates can be put in at SDC locations, in later stages of the fabrication process for a reduced fingerprinting cost.

A. Introduction to Satisfiability Don't Care Fingerprinting

In this chapter we propose a new methodology for creating fingerprints that rely on the concept of a Satisfiability Don't Care, a condition that occurs regularly in virtually every circuit of non-trivial size. This new method for generating fingerprints allows us to create a fingerprint that can be implemented in the later stages of the VLSI design cycle, thus bypassing the need for expensive redesign. Figure 14 shows a trivial example of this.



(a)

A	*B*	*C*	D	D'
0	0	1	1	1
0	1	1	0	0
1	0	1	1	1
1	1	0	1	1

(b)

Figure 14. Example SDC modification (a) Gate Replacement (b) Truth table for both circuits

This modification will not make a functional change at the primary outputs of the circuit, but it will make internal functionality different if tested. We also propose one possible encoding scheme that makes the fingerprint prohibitively difficult to modify without violating the encoding scheme or without being detected.

In attempting to create a robust, reliable fingerprint several challenges need to be addressed. First, we need to define where we can make modifications to our IC such that we can see differences at intermediate wires, but not at the primary outputs. The second challenge is to find a computationally inexpensive way to determine whether or not gates have SDC conditions without utilizing the deterministic exhaustive search usually associated with locating SDCs. Our final challenge is to make it difficult for adversaries to modify our fingerprints. If this is not done, adversaries can duplicate the circuit with valid fingerprints making it impossible for a developer to track their devices.

B. Satisfiability Don't Care Based Fingerprinting

In this section, we briefly discuss the concept behind SDCs and then how we utilize SDCs to create a fingerprint for IP protection. The goal of this work is to show how distinct fingerprinted circuits can be created effectively at logic synthesis level and how post-silicon techniques can help to improve the practicality of fingerprints that is lacking in the current literature [33, 3].

1. Satisfiability Don't Cares (SDCs)

Satisfiability Don't Cares are a Boolean concept used in circuit design optimization. Considering all the primary input (PI) signals, and the internal signals from each logic

gate in a circuit, SDC conditions describe the signal combinations that cannot occur. For example, consider the 2-input NAND gate from Figure 14, C=NAND(A,B), we cannot have {A=1,B=1,C=1}, {A=0,C=0}, or {B=0,C=0}. In general, for a signal *y* generated from logic gate $G(x_1,x_2, \ldots, x_k)$, the SDC at this gate can be computed by the equation

$$SDC = G(x_1, x_2, \ldots, x_k) \oplus y \qquad (1)$$

Which means that when the expected output *y* is at odds with the actual output of the gate function G and SDC condition has occured. In the example of the above 2-input NAND gate, the SDC conditions can be obtained from Equation (1) as follows:

$$\overline{AB} \oplus C = \overline{AB}\bar{C} + ABC$$

When some of these signals fan-in to the same gate later in the circuit, the SDC conditions can be used to optimize the design. In our approach, we will use these SDC conditions to embed fingerprints as illustrated in Figure 14.

2. SDC Based Fingerprinting

The main idea presented in this work is how SDCs can be utilized to create a fingerprint that is both robust against attack and unique enough to have one for each device created. By locating gates that have SDCs leading into them, which we refer to as *fingerprint locations*, and finding alternative gates, we can modify the circuit by using either the original gate or one of its alternatives at each fingerprint location, to generate different fingerprinted copies. We now analyze and solve the following SDC based fingerprint location problem:

Given an IP in the form of a gate level netlist, find a set of fingerprint locations, determine the alternative gates at each location, and define a fingerprint embedding scheme to create fingerprinted copies of the IP with any k-bit fingerprint.

Before presenting our solution to the problem, we list the necessary assumptions and define the terminologies.

A1. The given netlist should be sufficiently large to accommodate the k-bit fingerprint.

A2. The given netlist is optimized and does not have internal gates producing constant outputs. Circuits normally can be simplified if we replace constant-valued variables with their value (0 or 1).

A3. All primary inputs (PI) to the circuits are independent. If one PI depends on other PIs (e.g. the complementary variables in dual-rail logic), we can consider this PI as an internal signal.

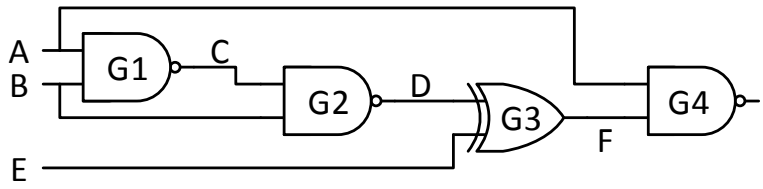


Figure 15. Example of a dependent line. B is a dependent line for gate G2 and A is a dependent line for G4.

For a given gate *g* in a circuit, a *cone rooted at gate g* is any sub-circuit that directly or indirectly produces a fan-in for gate *g*. A *dependent line/fan-in for gate g* is defined as a signal that directly or indirectly impact two or more of *gate g*'s fan-ins. For

example, in Figure 15, gates {G3,G4} is a cone rooted at G4 with inputs {A,D,E}; {G1,G2,G3,G4} is also one with inputs {A,B,E}. A is a dependent line for G4 and B is a dependent line for G2 (but not for G3 or G4).

A necessary condition for fingerprint locations: a gate must have dependent lines to be a fingerprint location.

When a gate, say G1 in Figure 15, does not have any dependent lines, its fan-ins will be independent and thus all possible fan-in combinations may happen. No SDC can be found. On the other hand, dependent lines do not guarantee a gate to be a fingerprint location. Consider the dependent line A for gate G4 in Figure 15, it is easy to see that when B=0, we have F=E', which is independent of A, so all four combinations of A and F can be fan-in to G4 and thus G4 cannot be a fingerprint location.

Based on the above observations, we propose the following heuristics to find fingerprint locations for k-bit fingerprints:

1. find a topological order of the gates G_1, G_2, …, G_n;
2. for each gate G_i (i=1, 2, …, n)
3. { if (G_i has a dependent line)
4. { find the cone rooted at G_i whose inputs are the dependent line or the intermediate lines closest to G_i;
5. for each combination of G_i's fan-ins that does not happen
6. { mark G_i as a fingerprint location;
7. record this fan-in combination;
8. update the number of fingerprint bits, FP;
9. if (FP > k)
10. { i = n+1; break;}
11. }}
12. mark G_i's output as PI;}

We search the gates for fingerprint locations following a topological order (Lines 1-2). If a fingerprint location is found, we mark the output of that gate as PI (Line 12).

In Line 3, we trace each fan-in of gate G_i back to PIs; whenever we see two fan-ins share the same signal, that signal is a dependent line. Then we construct the core rooted at G_i by backtracking each fan-in of G_i until we find the source of the dependent line or the closest, intermediate or primary, input signals to the cone for G_i that don't include the dependent lines (note here the PIs can either be the PI of the entire circuit of the fan-out of a fingerprint location as we mark in Line 12). Next we simulate all the combinations of input signals to this core and observe whether they can create any SDC at G_i's fan-in (Line 5). If so, we find a new fingerprint location in G_i and update the number of fingerprint bits (FP) we can produce (Lines 6-8). When FP becomes larger than k, the number of bits in the required fingerprint, we force the program to stop (Lines 9-10). The way to update FP depends on how fingerprint will be embedded, which we will discuss next.

Correctness of the heuristics: the heuristics may not find all the fingerprint locations, but the ones it finds as well as the SDC conditions (Line 7) are all valid.

This claim states that our heuristics will not report any false fingerprint location or SDC conditions. This ensures that when we do the gate replacement, the function of the original circuit will not be altered (requirement (1) for fingerprint). We will omit the proof of this claim due to space limitation.

Complexity of the heuristics: this is dominated by the size of the cone rooted at the gate under investigation. In Line 5 (other operations are either *O(1)* or *O(n)*), we have to solve the Boolean satisfiability to check whether each fan-in combination will occur or simply do an exhaustive search for all the combinations of the inputs to the core (which we choose to implement for this paper). In both cases, the

complexity will be exponential to the number of inputs to the core. However, after we consider the fan-outs of fingerprint locations also as PIs, our simulation shows that the average number of inputs to the cone is only 5.24. The heuristics' run time is in seconds for all the benchmarks.

3. Fingerprint Embedding Schemes

For each fingerprint location and its SDC conditions, we propose two replacement methods to embed the fingerprint:

R1. Replace the gate at the fingerprint location by another library gate where the two gates have different outputs only on the SDC conditions at the fingerprint location.

R2. Replace the gate at the fingerprint location by a multiplexer.

Figure 14 shows one example of **R1**, where a 2-input NAND gate and a 2-input XOR gate become inter-exchangeable when the input combination 00 is a SDC condition. Suppose that there are p_i different library gates (including G_i) which are can replace gate G_i, by choosing one of them, we can embed $\lfloor \log(p_i) \rfloor$ bits. So we update FP by this amount in Line 8 of our heuristics.

In **R2,** an m-input gate can be replaced with a 2^m x1 multiplexer (MUX). The selection lines of the MUX are tied to the original inputs of the gate and the data inputs are tied to either Vdd or Gnd, to match the patterns to implement the needed gate. Because the 2^m x1 MUX can realize any m-input function, if there are p SDC conditions at gate G_i, we can find 2^p gates as the alternative for G_i, including itself. So we will increase FP by p in Line 8 of our heuristics.

Option **R1** will require that new masks be created for each fingerprint, which is an expensive process. This leads us to prefer option **R2** which gives us the flexibility for post-silicon configuration. With this option we can utilize fuses, or other engineering changes such as the one presented in [36], to implement a fingerprint bit string in a circuit at the post-silicon phase. However this comes at the cost of high design overhead due to the large size and delay of the MUX. Figure 16 illustrates how the fuses will be used to implement the fingerprint.

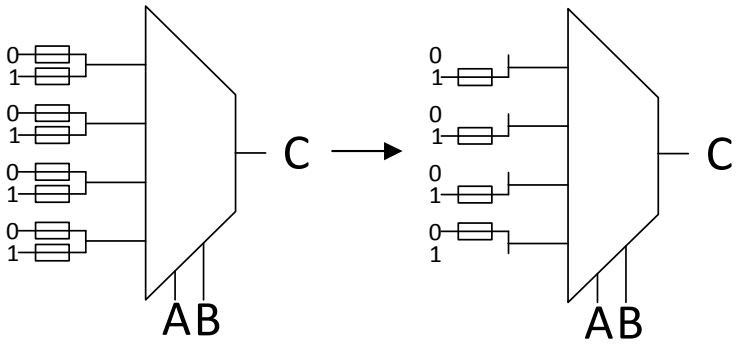


Figure 16. Multiplexer replacement technique. Left: An unconfigured MUX; Right: A MUX configured to run as a 2-input NAND gate.

C. Security Analysis

We first briefly discuss fingerprint detection because this is directly related to most attacks. When an adversary can detect the fingerprint, he may have an easier time to remove or change the fingerprint than with no knowledge about the fingerprint.

Fingerprint detection: when we are allowed to open up the chip and view its layout, we can recover the fingerprint by identifying the gate type at each fingerprint location (for **R1**) or checking the configuration at each MUX (for **R2**).

As we will show in the next section, there are abundant fingerprint locations in real-life circuits. Therefore we can choose to embed fingerprint bits (or part of it) at gates

that are visible to output pins. Then when we inject the SDC conditions to the fingerprint location, we can tell the gate type (and thus the fingerprint bit) from the output values. Consider Fig .1, if we inject B=0 and C=0, if we observe 1 as the value for D, we know the gate is a NAND; otherwise it is a XOR.

Now we consider the following attacking scenario based on the adversary's capabilities.

1. Simple Removal Attack

The most obvious attack against a fingerprint is to simply remove it. This requires that an adversary knows every location on an IC that our fingerprinting algorithm has modified, and more importantly, a way to remove these fingerprints without affecting the functionality of the original IC. In both **R1** and **R2**, because the fingerprint locations are required to provide the correct functionality of the circuit/IP, simply removing them will destroy the design and make the IP useless.

2. Simple Modification Attack

An adversary may also attempt to modify, instead of removing, a fingerprint to attempt to distribute additional copies of an IC that were not approved by the original developer. These copies will behave the same way as the original IC but will contain fingerprints that the original developer did not produce. To achieve this, an adversary can attempt to modify the fingerprint locations so that they create an unused bit string.

For **R1**, this is the same as modifying the gate replacement based watermarks, which is known to be hard [1]. For **R2**, the attacker can find the fingerprint locations by

looking for the MUXs. He can try to change the configuration of these MUXs, but without fully reverse engineering the design, it will be hard to maintain the correct functionality.

Finally, we mention that our results show that we can easily embed hundreds and thousands of fingerprint bits. So we will have room to choose fingerprints that are relatively far from each other (e.g. in terms of Hamming distance). Then the attacker has to remove a large amount of fingerprint bits to remove the fingerprint.

3. Collusion Attack

This is a more attack when multiple adversaries or an adversary with access to multiple fingerprinted IPs collude by comparing their copies to find the fingerprint locations and the alternatives at each location. This can be used to attack both **R1** and **R2**.

To prevent this, we propose that the fingerprint bit string to be chosen using certain encoding scheme (such as error correction or any coding designed for integrity checking) such that these bits will become dependent and have certain pattern, property, or structure. In this way, the colluded bits will fail to have these required pattern/property/structure.

D. Simulation Results and Discussion

To validate our proposed SDC-based fingerprinting approach, we first see how many fingerprint bits we could hide in a circuit, and second, we want to know how much design overhead we would have from embedding the fingerprints.

To accomplish this, a program was written in C++ to find all possible SDC locations and replace them with multiplexers, following the methodology presented in section III. This was run on a number of circuit benchmarks from the ISCAS-85 [37], ISCAS-89 [38], and MCNC [39] benchmark libraries, seen under the Unmodified Circuit Information section of Table 6. The library used for delay and area measurements is the Oklahoma State University standard cell library based on the TSMC 0.35μm technology [40]. Measurements for area and delay were collected using the program ABC [35].

1. Fingerprint Potential

The maximum size of the fingerprints we were able to find for the replacement methods R1 and R2 are given in Table 6. The number of fingerprint locations for R2 is also listed. R1 did not have more than one modification per location so the number of bits is equivalent to the number of fingerprint locations.

Table 6 also shows that the size of the original circuit to be an indicator of how large a fingerprint can be produced. As expected, smaller circuits such as c432, c880, etc. were only able to create short fingerprints for the R1 replacement method. We believe this is acceptable because a circuit of this size could be reverse engineered easily, and any security features could be removed. Despite this, every **R2** replacement and most **R1** replacements could create a sufficient number of fingerprints. For example, **R2** can find locations for more than 100 fingerprint bits for all but 3 small circuits. For the 3 largest ones, we have more than 1000 fingerprint bits, enabling us to create more than 2^{1000} fingerprinted copies.

The number of inputs to the rooted cones has an average count of 5.24. Table 6 also shows that there is no correlation between the size of the circuit being processed and the number of inputs to the rooted cone. As stated before, this means that the runtime complexity does not go up exponentially with the size of the circuit. This explains the fact that we have runtimes in the range of seconds for every circuit tested.

*Table 6. Fingerprinting Results Using Both the **R1** and **R2** Gate Replacement Techniques*

	Unmodified Circuit Information				*R1*		*R2*			
Circuit	*Gate Count*	*Area*	*Delay*	*Average # of inputs to rooted cones*	*Maximum Bits Found*	*Area Overhead*	*Modification Locations*	*Max Bits Found*	*Max Size Fingerprint Area Overhead*	*32 Bit Fingerprint Area Overhead*
C432	183	456	3.94	3.86	3	0.00%	12	48	80.00%	53.33%
C880	304	797.6	3.19	5.16	12	5.12%	38	90	117.05%	39.32%
C1908	398	1068	4.73	4.02	21	6.29%	53	118	113.86%	39.03%
C499	454	1164.8	3.1	4.27	48	13.60%	57	74	72.80%	37.23%
vda	712	2026.4	1.75	7.91	57	11.33%	104	296	113.46%	12.36%
C3540	855	2357.6	4.91	4.83	24	3.02%	161	522	232.41%	15.30%
dalu	960	2638.4	5.19	4.21	83	9.70%	128	381	94.85%	10.61%
t481	1092	3382.4	2.14	8.67	16	1.73%	52	229	51.61%	8.73%
k2	1383	3732	2.69	8.75	69	7.46%	136	597	212.09%	31.40%
s9234	1478	4051.2	4.35	4.72	32	2.37%	139	659	124.47%	7.29%
i8	1638	4492.8	1.97	4.29	6	0.59%	315	650	159.24%	9.15%
i10	1944	5200.8	5.94	5.44	59	4.00%	257	832	210.23%	6.20%
C6288	2303	6210.4	15.18	3.14	213	11.50%	379	551	122.21%	6.18%
s13207	2471	6824.8	4.23	5.67	27	1.17%	106	555	59.91%	3.88%
s15850	2765	7689.6	5.66	3.97	45	1.89%	221	590	74.05%	3.63%
des	3629	9716	2.82	4.86	324	10.66%	450	1000+	66.06%	3.58%
s38417	8122	21745.6	3.38	5.88	163	2.63%	412	1000+	44.87%	1.85%
s38584	9447	25536.8	3.65	4.66	114	1.35%	363	1000+	36.80%	1.06%
Average:				5.24		5.25%			110.33%	16.12%

2. Design Overhead

Overhead is a major concern with IP security techniques. There will always be a trade-off between circuit performance and security because security features always need to be built on top of the original optimized ICs.

Table 6 shows that the average area overhead for **R1** is only 5.25%, which is acceptable. This is because **R1** only replaces individual gates with another gate with area and delay on the same order of magnitude.

For small circuits, the overhead for **R2**, is expected to be significant. This is because we are substituting standard logic gates, such as AND, NAND, etc., with multiplexers which are significantly larger and slower. As stated above, the overhead will be mitigated by the size of the circuit. In Figure 17 we can see a general downward trend for both delay and area overhead for circuits with a 32-bit fingerprint implemented. We chose a 32-bit fingerprint because the likelihood of needing 2^{32} different fingerprints to mark each manufactured circuit, of a specific design, is low, and all of the benchmarks we worked with could handle this many fingerprints. Currently, we are working on performance-driven methods to balance fingerprint size and overhead (see section VI).

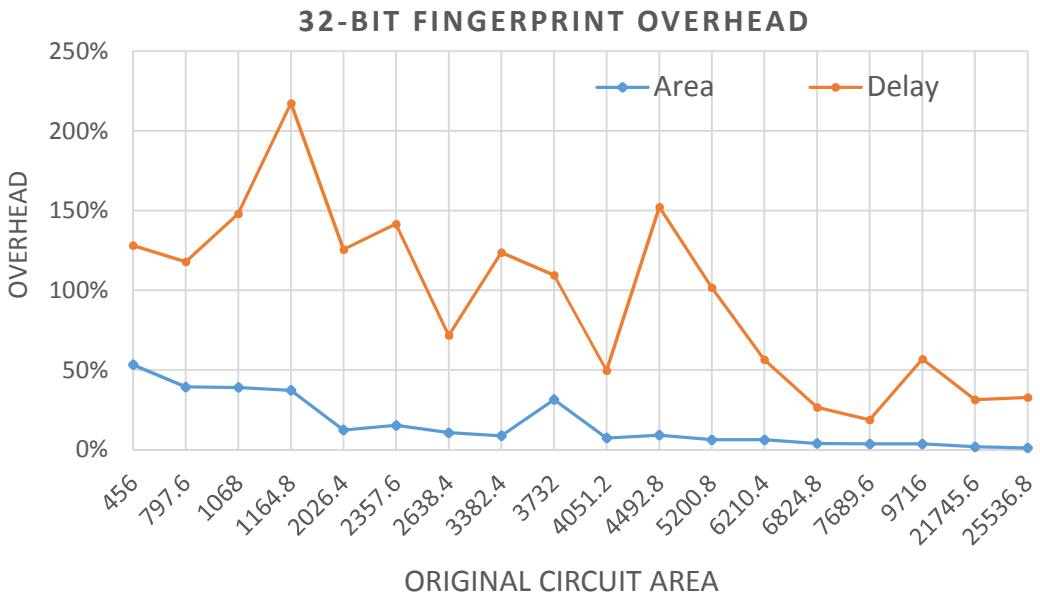


Figure 17. Overhead results for 32-bit fingerprint using multiplexer replacement.

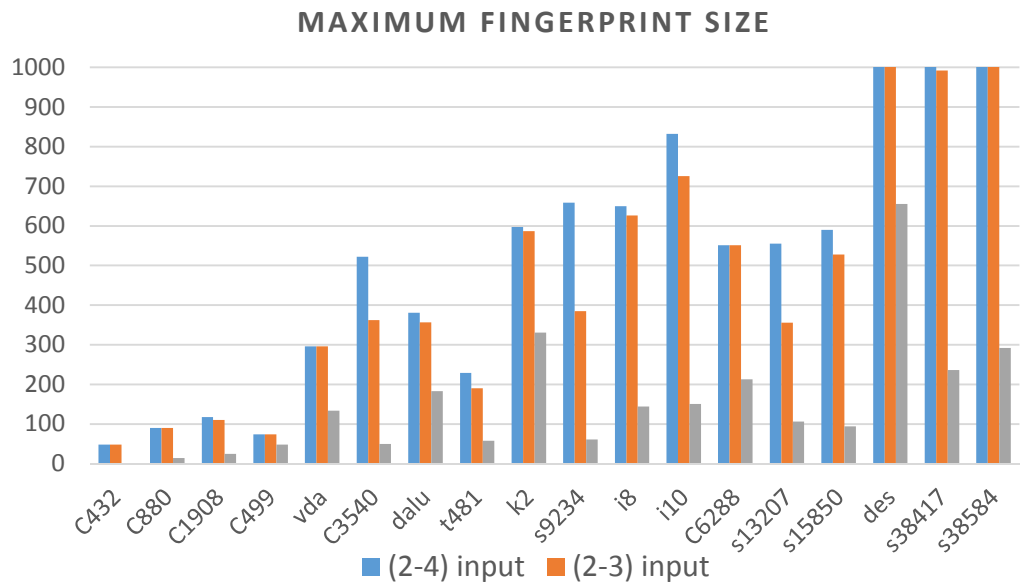


Figure 18. Number of bits found for a 1024 bit fingerprint with gate replacement limitations

3. **R2** Gate Size Replacement Constraints

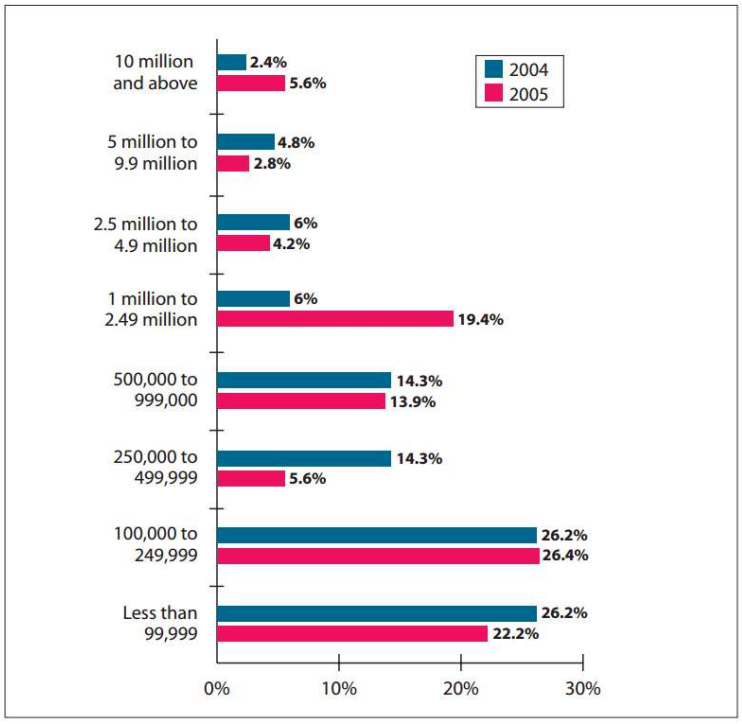
Finally, we wanted to determine what would happen if we constrained ourselves to replacing gates that took a specific number of inputs. We ran three tests based on our cell library using the **R2** replacement method. The first test replaced gates that had between 2 and 4 inputs, the second test focused on 2 or 3 inputs, and finally the third test replaced gates with only 2 inputs. The number of bits that were able to be found are shown in Figure 18.

Limiting the types of gates we replace caused a significant reduction in the sizes of the fingerprints we could produce for most of the circuits, especially when we limited ourselves to only replacing 2 input gates. In addition, removing 4-input gates does not improve our overhead significantly. We found that the average area overhead improvement is on average 10%. The library we used only had access to 2-to-1 multiplexers, so we had to implement MUXs of 4x1, 8x1 and 16x1 with these 2x1

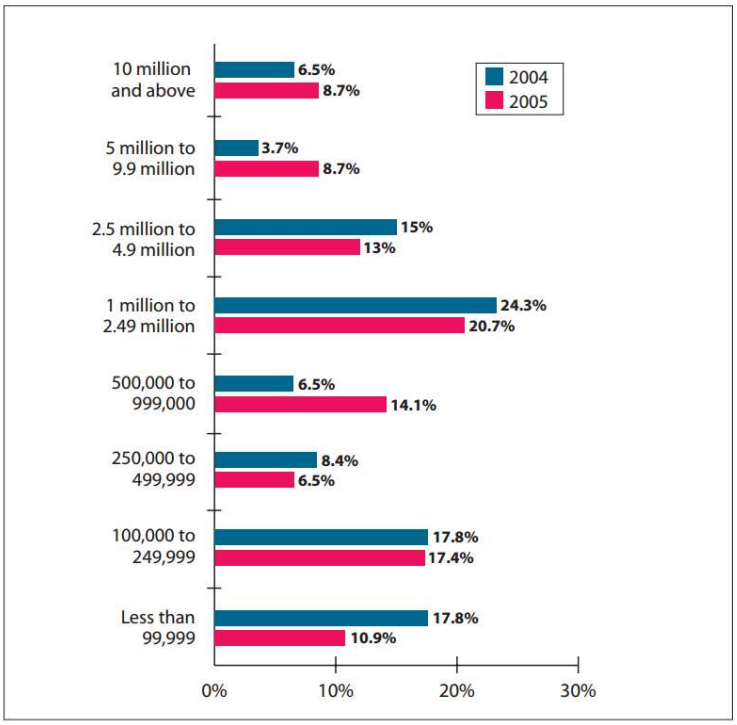
MUXs. However we expect that with customer designed 4x1, 8x1, or 16x1 MUXs, area and delay overhead will drop.

4. Other overhead considerations

There is another important fact to consider when looking at these experimental results. The tests run were done on circuits that had at most tens of thousands of gates, and many had much less. A fingerprint's size does not scale with the size of the circuit, only with the amount of security that needs to be provided and the number of circuits that need a unique fingerprint. This fingerprinting method only makes changes to the gate level design and thus, a n-bit change to a 500 gate circuit will result in a similar overhead to that of a 5,000,000 gate circuit, but in the latter it will be a much smaller percentage overhead. Figure 19 shows the sizes of the ASICs produced in both Taiwan and China in 2004 and 2005, and as can be seen, approximately 75% of the circuits have a size greater than 100,000 gates. This will only trend upwards with time, as process technology improves. This means that even with the data in Figure 17, most production circuits with a 32-bit fingerprint, would have, at least, approximately ¼ the area overhead and a similarly reduced delay overhead when compared to the largest circuit we simulated.



(a)



(b)

Figure 19. From 2005, (a) ASIC gate counts in Taiwan (b) ASIC gate counts in China [41]

E. Conclusion

We propose a novel fingerprint approach for IP protection based on the well-known concept of Satisfiability Don't Care (SDC) conditions. Utilizing the SDCs in the circuit, we find alternatives to replace a gate, which do not change the functionality, to generate fingerprints. More importantly, we propose to use configurable cells (such as multiplexers) as the replacement, which allows us to do the configuration based on fingerprint at the post-silicon phase and solves one of the most challenging problems for fingerprinting.

This approach is promising, as validated by the simulations where we embedded hundreds or thousands of fingerprint bits successfully. The high design overhead is caused in large part by our current algorithm, which does not consider performance. Some of the overhead can be easily reduced or controlled given the large selection pool of fingerprint locations. For example, we can avoid picking gates from critical path to reduce delay penalty, or use smaller replacement options to save area; we can also use customer designed 4x1, or 8x1 MUXs. Currently, we are working on these performance driven methods.

CHAPTER 5: FINGERPRINTING THROUGH FINITE STATE MACHINE MANIPULATION

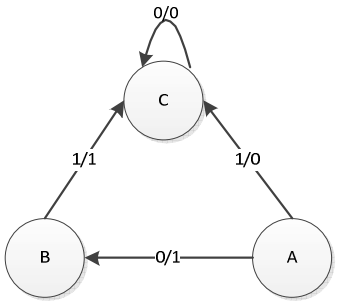
The third fingerprinting method we developed for this dissertation was based off of our previous work on finite state machine (FSM) security, which can be found in Appendix A. This method utilizes the unused transitions between states in an incompletely specified FSM to create fingerprints for circuits. In this chapter, the terms transition, path, and edge are used interchangeably to represent the connection between two states in an FSM, specified with a specific input, starting state, ending state, and output.

A. Methodology of the work in Appendix A

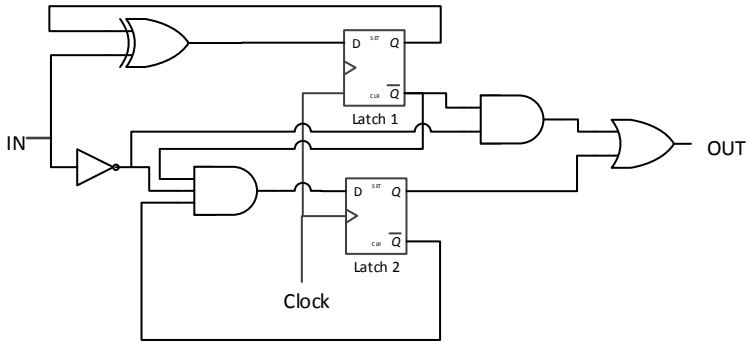
Before describing the work we propose: creating a fingerprint based on changing the flip flops in a finite state machine; we will give a brief synopsis of the work this idea was based off of.

The goal of the work in Appendix A was to prevent either the misuse of unspecified states and transitions in sequential circuits that were not fully specified. Figure 20 gives us a simple example to show this point. In Figure 20(a) we can see an example FSM with three states and their paths of the format input/output. We can observe that the FSM is not fully specified because we do not have a power of 2 states, which would be a minimum of 4, and not all of the paths are specified, each state should have a path for an input of 0 or 1. Figure 20(b) shows the optimized circuit design for the FSM in Figure 20(a), and Figure 20(c) shows the behavior of the circuit design. It can be observed in Figure 20(c) that there are three new paths that have been specified and a new state as well.

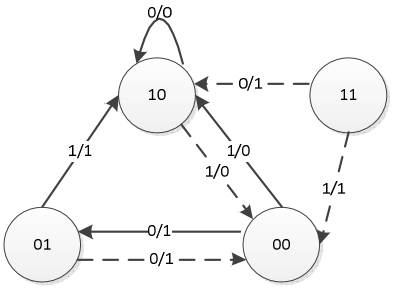
The problem that this shows, is that if we originally wanted state A (Figure 20(a)) or 00 (Figure 20(c)) to remain inaccessible once it was left, that was lost once the circuit design was created. This can be prevented by fully specifying the rest of the circuit, but this can be expensive as well. In addition, it is also possible for adversaries who have access to the device, such as a foundry or design firm, to add paths to states that should not have paths to them by utilizing unspecified paths.



(a) The original 3-state FSM as the system specification.



(b) The logic/circuit implementation of the 3-state FSM shown in (a).



(c) The 4-state FSM generated from the circuit shown in (b).

Figure 20. The illustrative example. States A, B, and C in (a) correspond to the states 00, 01, and 10, respectively in (c).

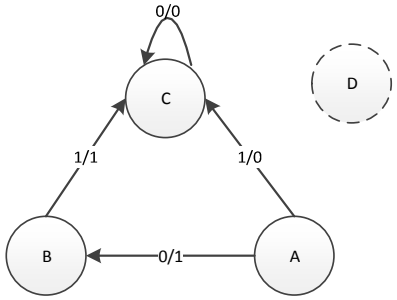
In the work in Appendix A, we considered a trusted FSM to be a device that could not access "safe states" (states that were not meant to be accessed), once the "safe states" were left, like in Figure 7.

B. FSM Manipulation Based Fingerprints

The goal of this work was to explore the possibility of creating a practical IC fingerprinting method, for sequential circuits, based on the work done in Appendix A. It was realized that most circuits that are designed are not fully specified because it allows for optimization software to reduce performance overhead, giving faster speeds, lower power usage, and smaller dies. As stated before, this is the difference between Figure 20(a) and (c). What was found in Appendix A though, was that optimization software, in this case SIS, does not always give a minimally sized circuit. In section 4.2 of Appendix A, the attacker randomly added a transition to the FSM in question and in many cases the performance overhead was negative, meaning that the new circuit actually performed better than the original. This result led to the

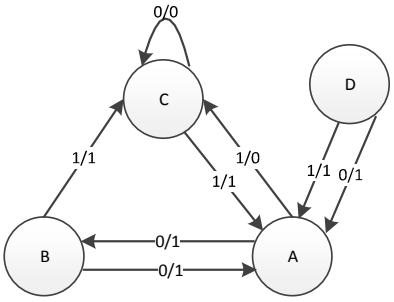
belief that there was a potential to modify the circuit while still in the FSM design stage and create fingerprints with an acceptable performance overhead.

As a quick motivational example we present Figure 21 and Figure 22. Figure 21 shows the FSM from the previous section with its transition table. There are two missing transitions from the states that are specified, and one missing state with two more transitions that are missing as well. If this FSM was mapped, using any technology library, each of these transitions would be specified regardless of the designer's original intent.



Input	Start State	End State	Output
0	A	B	1
1	A	C	0
0	B	-	-
1	B	C	1
0	C	C	0
1	C	-	-
0	D	-	-
1	D	-	-

Figure 21. Incompletely specified FSM



Input	Start State	End State	Output
0	A	B	1
1	A	C	0
0	B	A	1
1	B	C	1
0	C	C	0
1	C	A	1
0	D	A	1
1	D	A	1

Figure 22. One of numerous possible completely specified FSMs

Figure 22 is one example of how the unspecified transitions and states in Figure 21 could be used to create a fingerprint. In this case, each unused transition was given

an output of 1 and sent to state A. This is an unlikely full specification that would occur from mapping the FSM in Figure 21 so the chance of collision with another non-fingerprinted circuit is unlikely. As a trivial example, we have 256 different FSM configurations, for the FSM in Figure 21, to fully specify the design. This example shows that while not all options have a great performance, there are so many of them that it does not make a difference.

Although this work focused on adding unused transitions to an FSM there were several challenges to overcome. First we needed to determine the unused transitions and states in each FSM. If there were no unused transitions or states we needed to modify the FSM further to allow for a fingerprint. The second challenge was determining how many fingerprints we could create and how many we should test. In most cases, there are an astronomical number of potential FSM configurations using the unused transitions and states from an incompletely specified FSM. We needed to reduce this to be able to test properly. Finally, as with the rest of our fingerprinting methods, we needed to maintain a low overhead so that the fingerprint would not be too costly.

1. Determining Unused Transitions and States

Every FSM that is created using IC technology will have a state count that is equal to a power of two, and the same can be said for the number of inputs and outputs for each state as well. Many FSMs that are produced will only use a subset of the possible states, inputs, or outputs leaving an FSM incompletely specified. As a result, a number of transitions will also go unused as they are a function of how many inputs and states are utilized. This is where the focus of our work lay in Appendix A. In

that work, we saw these unspecified aspects of the FSM as potential threats to sensitive data or functionality, but in this work they are to be considered flexibilities that allow us to make small changes in the circuit that, while not changing the original functionality, will allow us to create fingerprints for tracking ICs

2. Potential Fingerprint Count

The major advantage of using FSM manipulation to fingerprint circuits is that there is a potential for a huge number of fingerprints given the correct conditions. For every circuit that is missing n transitions and has m states, there are m^n potential different combinations to fully specify the circuit. This is because for every missing transition at state *X* can go from state *X* to any other state in the FSM, including itself. Unfortunately, this does not mean that every transition can or should be used, and in many cases, fully specifying an FSM that is missing a large number of transitions, will cause the performance overhead to climb drastically.

$$Possible\ Fingerprints = \binom{e_{missing}}{n} * {s_{total}}^n \quad (2)$$

As an alternative, this work utilized both single and double transition additions to our circuit. To determine the number of possible fingerprints that could be generated from each of these we used equation (2), above. In this equation *n* represents the number of transitions we want to specify or insert to our FSM, $e_{missing}$ is the number of missing transitions in our FSM, and s_{total} is the maximum number of states in the fully specified FSM we are working with.

3. Maintaining Overhead

One concern when any sort of functionality to a circuit is how it will affect its performance. A circuit may have perfect security, but if it costs 100 times more to produce than a circuit with no security, it will have very few reasonable applications. Hence, it was necessary to make sure that when we add fingerprints to these FSMs so that once they were mapped to a technology library, the overhead compared to the original mapped FSM was not significant.

To prevent significant overhead, we can simply map each change we make to an FSM and compare the results to the original FSM's mapped circuit. From here we can set a threshold on the various performance metrics and throw out any FSM modifications that cause undue overhead. Although, it was not tested in this work, when generating fingerprints that utilize more than one transition, it will likely be faster to determine the FSM with the lowest overhead for a single transition, then use those to add new transitions, one at a time and winnow out the transitions that cause large overhead.

4. Security Analysis

A major part of any security technique is an analysis of its strengths and weaknesses. FSM manipulation fingerprinting is an extremely robust security device because it is implemented in the upper levels of the VLSI design cycle, even further above than the work in [3] and [33].

It is important to note that assumptions must be made in regards to security and ICs because of the limited nature of their size and functionality. An adversary with significant resources and time can reverse engineer any device and figure out its core functionality, but in most cases this would be more costly than designing the circuit from scratch, so it is assumed that the adversary will not take this approach. In this

work, the assumption is made that adversaries would attempt to remove or modify the fingerprint by themselves, or with another adversary in a collusion attack.

a) Simple Removal Attack

The first attack that is considered, is one where an adversary simply tries to remove the fingerprint from the circuit, without adverse effects to the device's functionality. Unlike the SDC and ODC fingerprinting methods, this fingerprinting method is implemented in the upper levels of the VLSI design process so the actual fingerprint data is almost inextricably linked with the functionality.

For an adversary to completely remove the fingerprint they would need to reverse engineer the device until they can determine the actual FSM that the work was based on. Even at this point, an adversary would need to have an intimate knowledge of the device's functionality so that they do not remove transitions that were vital for functionality. For these reasons, it is highly unlikely, or extremely expensive for an adversary to remove a fingerprint from the circuit.

b) Simple Modification Attack

The other major concern with fingerprinting ICs is that an adversary will attempt to change a fingerprint to one that is not in use or one that another customer has, thus thwarting attempts to track the adversary's malicious behavior. In order to do this well, the adversary would, again, need to be able to reverse engineer the circuit back to the FSM. At this point if they can determine which transitions define the fingerprint in their device they can change this then remap the circuit. As with the

previous attack, this is an extremely cost inefficient process and does not pose a major threat.

The other major threat that the adversary can perpetrate would be making small modifications to the circuit or arbitrarily changing transitions that they do not believe to be necessary to the functionality of the circuit. This would also require knowledge of the functionality of the circuit to insure that the circuit is not rendered unstable or useless. If the adversary manages to do this though, the layout of the circuit would, with high probability, no longer be the same as one of the fingerprinted circuits. In this case a destructive analysis can prove that a circuit is a counterfeit and not reliable.

c) Collusion Attack

The final attack that is considered, is when two adversaries work together to determine the differences in their circuitry. Because fingerprinting is based on the individuality of every chip, this can expose a lot about the fingerprint, but with FSM manipulation, it is unlikely that adversaries would be able to counterfeit a fingerprint that another customer received.

If two adversaries colluded, they would be able to see where certain functionality of their circuits differed by doing an exhaustive run of all possible input combinations for every state, or by doing a full destructive analysis of their two circuits, both of which are expensive in terms of time and/or resources. This would give them an idea of what transitions would be part of their fingerprints, but not necessarily all of it. In addition, it would not tell them how other circuits implement other fingerprints because each circuit was mapped differently. Again, at best the adversaries could

make adjustments to hide the original fingerprint, at great expense, but an analysis of the circuit would show it to be counterfeit, and thus unreliable.

C. Experimental Setup

A number of experiments were run to prove the usefulness of this method. A program was written in PERL to manipulate a series of FSMs and provide the performance statistics for a number of uniquely fingerprinted circuits. The FSMs came from the MCNC and ISCAS '89 benchmark libraries, in the format of KISS2 files. KISS2 is a file format that simply states the number of inputs, outputs, states, and paths and then specifies what each path looks like in the format of:

<input> <current state> <next state> <output>

1. Circuit Characterization

The first task in these experiments was to characterize each benchmark circuit by recording its performance statistics (area, max negative slack, sum of negative slack, and power). Then each path in the circuit was recorded as well as the number of states. From this it could be determined how many possible states could exist in the fully specified FSM by rounding up to the nearest power of two. The same was done with the number of inputs. This allowed us to determine the number of missing paths for a fully specified FSM and what input patterns for which states they belonged.

2. FSM Manipulation

The second task for this experiment was to inject extra paths into our benchmark circuit FSMs and record the results. For each benchmark we went through every possible input combination, given the number of input bits the device started with,

and checked if each state was already assigned a transition for that input pattern. If that input pattern did not exist for that state, then we sequentially went through every possible transition that that state could make and tested for the area, maximum negative slack, sum of negative slack, and power usage. Figure 23 shows a high-level pseudo code implementation of this.

```
Input: state list, edge list, input bit length: inputLength, benchmark
Output: List of circuit performance stats with edges injected
Variables: x, y are state variables, i is an input variable copyBenchmark is a copy of the original benchmark file
Methods: edgeExists checks for the existence of the edge in the FSM, addEdge adds a new edge to the benchmark
file, circuitPerformance returns performance metrics
1)  for each state x:
2)      for i = 0 to 2^inputLength-1:
3)          if(edgeExists(x, i) == false)
4)              for each state y:
5)                  copyBenchmark = originalBenchmarkFile ;
6)                  addEdge(copyBenchmark, new Edge(i, x, y);
7)                  results += circuitPerformance(copyBenchmark);
8)                  delete(copyBenchmark);
```

Figure 23. Pseudo-code of FSM edge injection

This code was modified later to add two unique unused transitions from the list of unused edges. We had to make sure not to test the same FSM additions repeatedly and as such, the pseudocode below, in Figure 24 was used to iterate through all possible combinations of two transition insertions.

```
Input: state list, edge list, input bit length: inputLength, benchmark
Output: List of circuit performance stats with edges injected
Variables: u, v, w, and x are state variables, i and j are input variables copyBenchmark is a copy of the original
benchmark file
Methods: edgeExists checks for the existence of the edge in the FSM, addEdge adds a new edge to the benchmark file,
circuitPerformance returns performance metrics
1)  for each state u:
2)      for I = 0 to 2^inputLength-1:
3)          if(edgeExists(u, i) == false)
4)              for each state v >= u:
5)                  for j = 0 to 2^inputLength-1:
6)                      if( !(j <= i && v == u) && edgeExists(v, j))
7)                          for each state w:
8)                              for each state x:
9)                                  copyBenchmark = originalBenchmarkFile ;
10)                                 addEdge(copyBenchmark, new Edge(I, u, w);
11)                                 addEdge(copyBenchmark, new Edge(j, v, x);
12)                                 results += circuitPerformance(copyBenchmark);
13)                                 delete(copyBenchmark);
```

Figure 24. Pseudo-code for double FSM edge insertion

For many of these circuits, the number of missing edges was extremely large and took a long time to run. As a result, we did a random sampling of the possible fingerprints for each of the different benchmarks to get an idea of what the average results would be. We were able to implement this by implementing code similar to the pseudo-code in Figure 25.

```
Input: state list, edge list, input bit length: inputLength, benchmark
Output: List of circuit performance stats with edges injected
Variables: x, y are state variables, i is an input variable copyBenchmark is a copy of the original benchmark file,
count is used to count to our sample size
Methods: edgeExists checks for the existence of the edge in the FSM, addEdge adds a new edge to the benchmark
file, circuitPerformance returns performance metrics, and randomState gets a random state from a list of states,
deleteEdge removes and edge from a list of edges
1)  for each state x:
2)      for I = 0 to 2^inputLength-1:
3)          if(edgeExists(x, i) == false)
4)              add edge(x,i) to unusedEdges
5)  count = 0
6)  while(count < sampleSize)
7)      (x, i) = randomEdge(unusedEdges)
8)      y = randomState(state)
9)      copyBenchmark = originalBenchmarkFile ;
10)     addEdge(copyBenchmark, new Edge(i, x, y);
11)     results += circuitPerformance(copyBenchmark);
9)      delete(copyBenchmark);
10)     deleteEdge(unusedEdges, new Edge(x,i))
11)     count++
```

Figure 25. Random sampling FSM insertions pseudo-code

This code first determines which edges are not in use by the original FSM and stores them in a data structure. Then at random an edge is chosen and implemented in the same fashion as the other pseudocode. This code is run until we have reached the sample size we require. For the double insertion, the code is similar, except that we pull two random edges, and make sure to check that they are not the same, or that that combination of edges has not been implemented before.

D. Results

For this work we first present Table 7 with the base statistics about each of the benchmark FSMs that were modified. Table 7 gives the benchmark names, number

of input and output bits, the maximum number of states after full specification, current number of edges, the maximum number of edges after full specification, the number of missing edges in columns 1-7, respectively. Columns 8-10 represent the number of possible fingerprints that can be specified by inserting 1, 2 or all missing transitions in our benchmark FSM. As can be seen, this number grows by the equation in section B.2 and for even our smaller circuits, the number of possible fingerprints, based on column 10, is significant.

Table 7. Benchmark circuit base statistics

Name	*Input*	*Output*	*Max # of States*	*Edges*	*Max # of Edges*	*Missing Edges*	*# of Single Additions*	*# of Double Additions*	*Log2(Possible Fingerprints)*
bbara	4	2	16	160	256	96	1536	1167360	384
bbsse	7	7	16	1856	2048	192	3072	4694016	768
beecount	3	4	8	51	64	13	104	4992	39
cse	7	7	16	2028	2048	20	320	48640	80
dk27	1	2	8	14	16	2	16	64	6
dk512	1	3	16	30	32	2	32	256	8
ex1	9	19	32	7552	16384	8832	282624	3.9934E+10	44160
ex2	2	2	32	72	128	56	1792	1576960	280
ex3	2	2	16	36	64	28	448	96768	112
ex4	6	9	16	448	1024	576	9216	42393600	2304
ex5	2	2	16	32	64	32	512	126976	128
ex7	2	2	16	36	64	28	448	96768	112
keyb	7	2	32	2432	4096	1664	53248	1416822784	8320
lion9	2	1	16	25	64	39	624	189696	156
planet	7	19	64	6144	8192	2048	131072	8585740288	12288
planet1	7	19	64	6144	8192	2048	131072	8585740288	12288
pma	8	8	32	2928	8192	5264	168448	1.4185E+10	26320
s1	8	6	32	5120	8192	3072	98304	4830265344	15360
s208	11	2	32	36864	65536	28672	917504	4.2089E+11	143360
s27	4	1	8	64	128	64	512	129024	192
s386	7	7	16	1664	2048	384	6144	18825216	1536
s420	19	2	32	9437184	16777216	7340032	234881024	2.7585E+16	36700160
sand	11	9	32	64576	65536	960	30720	471367680	4800
sse	7	7	16	1856	2048	192	3072	4694016	768
styr	9	10	32	15344	16384	1040	33280	553246720	5200
tma	7	6	32	692	4096	3404	108928	5930911744	17020
train11	2	1	16	25	64	39	624	189696	156
train4	2	1	4	14	16	2	8	16	4

The area, power, maximum negative slack, and sum of negative slack was recorded for every manipulated FSM after being mapped. In addition to recording the performance metrics, for every benchmark that was tested, either for single or double transition insertion, we checked the overhead in comparison to the original

benchmark FSM. With this information, it was determined how many of the new fingerprinted FSMs met a certain threshold of 5% overhead for each performance metric, independently. It was also recorded how many of the fingerprinted FSMs met a threshold of 10% overhead for every metric.

1. Single Edge Insertion

For the first experiment, as stated before, a single new transition was inserted into the benchmark FSMs. For each benchmark, we took a sample size of 1500 unique, random unused transitions were inserted, unless the FSM had less than 1500 unused transitions, in which case everyone was tested.

The first major result collected, can be seen in Figure 26. Figure 26 is a chart depicting the average overhead of all of the sampled fingerprinted FSMs, separated by metric, and the benchmark. The important information here mostly comes from the general low overhead for each metric. In most cases, the overhead is not greater than 20% and in some cases is negative, representing an improvement over the original FSM, similar to the results found in the work in Appendix A. The only major discrepancy was for the circuit lion9. In this case, the circuit was extremely small, so fully specifying the circuit did not leave any room for optimization the way it did for the rest of the circuits. This low overhead is extremely important for security techniques because without it, most customers will not want to purchase the device.

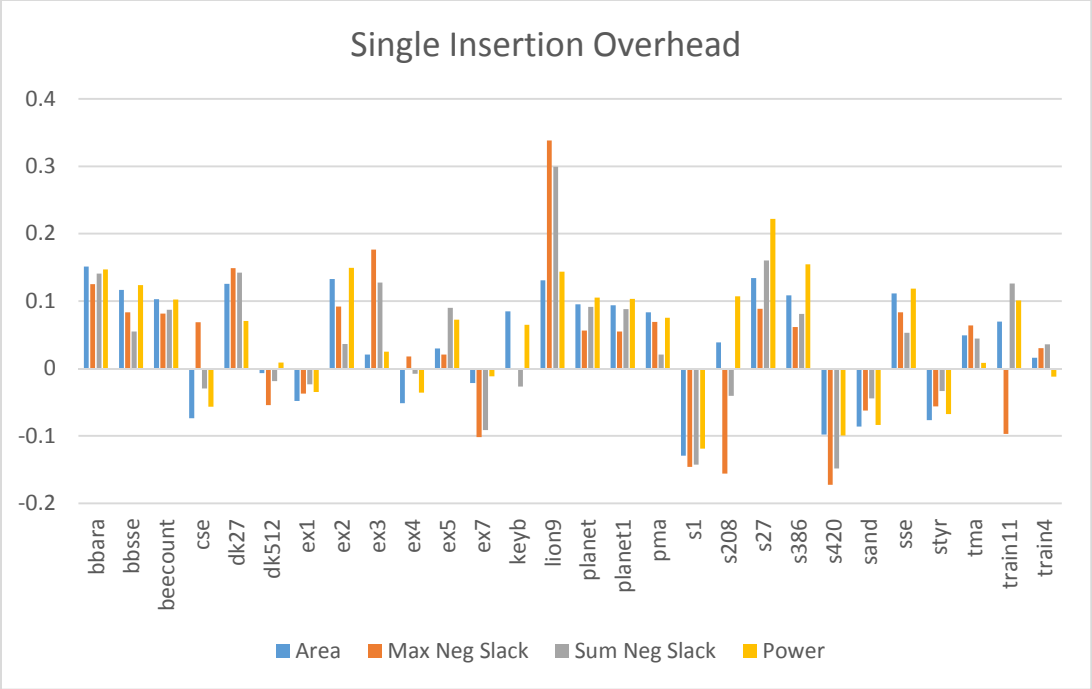


Figure 26. Overhead for each performance metric (Single Insertion Fingerprinting)

2. Double Edge Insertion

For the second experiment, a sample size of 1500 was used to get an idea of the performance of adding two new transitions to the benchmark FSMs. Again the same rule applied, where if there were not 1500 combinations of unused transitions, every possible unused transition was tested.

The first result for the second experiment, the double transition insertion is the same as for the single. Figure 27 is a chart showing the overhead for every performance metric, across all benchmarks. Here again, it is seen that the performance overhead for most of these FSMs is below 20% with a few still in the negative range. The FSM for lion9 again gave us a more extreme overhead, but for the same explanation as in the previous section. The biggest difference between the single and double insertion

was that the double insertion had slightly elevated overhead, which is to be expected for restricting the optimization space.

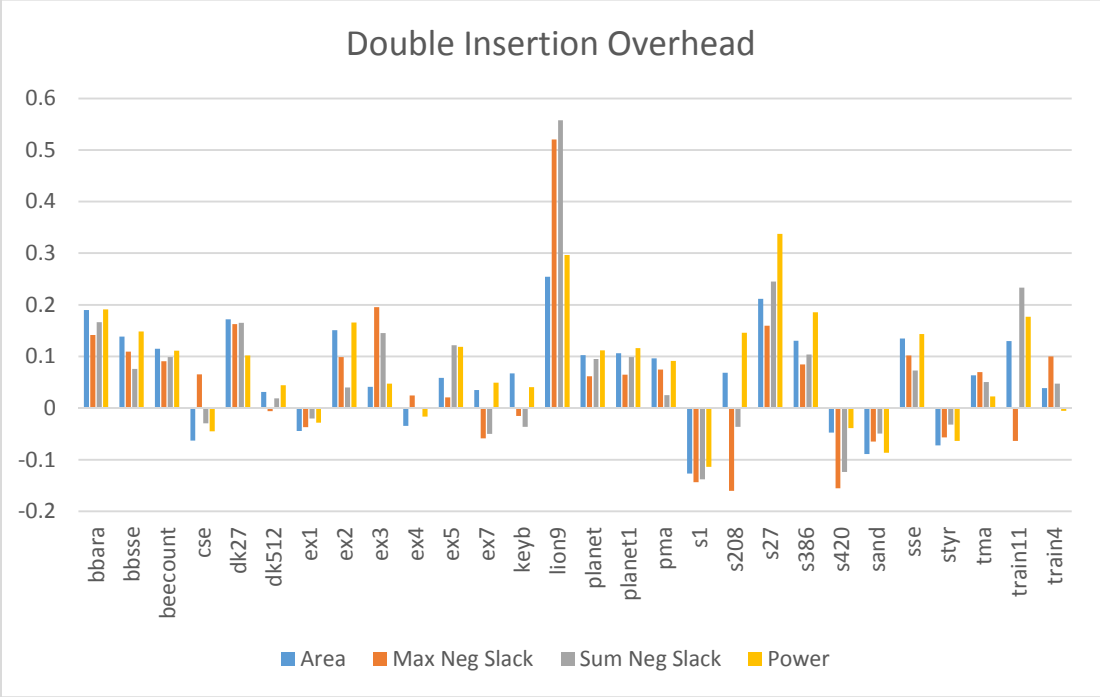


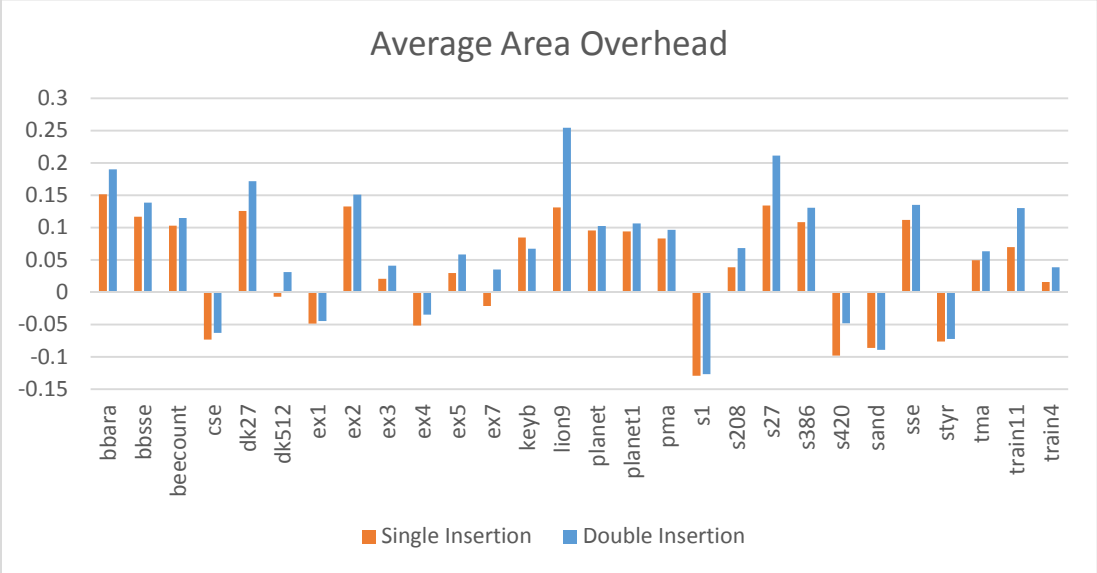
Figure 27. Overhead for each performance metric (Double Insertion Fingerprinting)

3. Comparison of two techniques

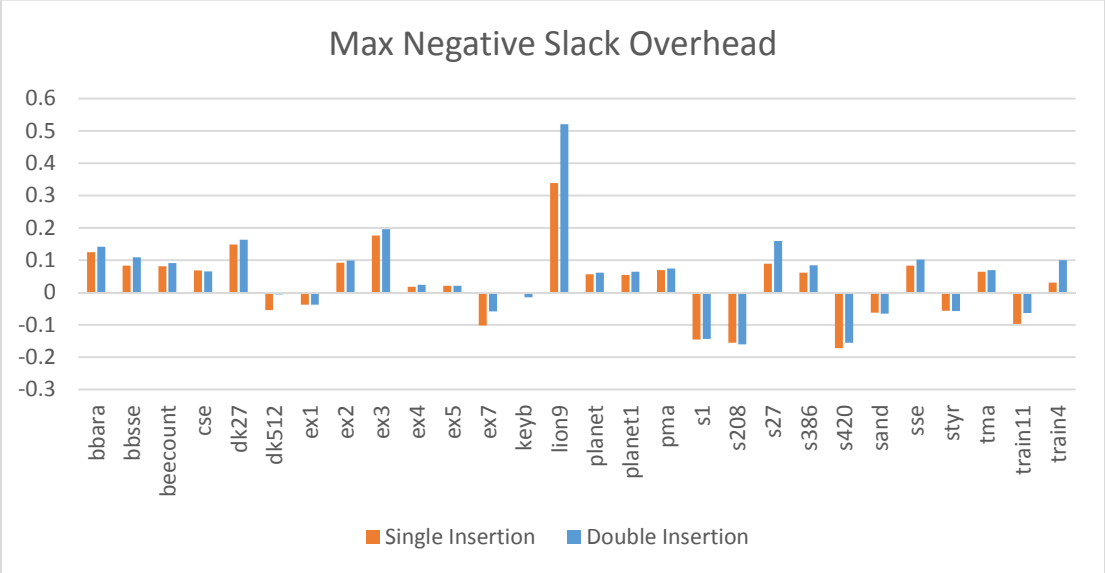
One of the more important pieces of information gathered in this work was the direct comparison between the single insertion and double insertion FSM manipulations. From Table 7, it is known that the double insertion method would have a larger number of potential fingerprints, but if the overhead was significantly higher or the number of acceptable fingerprints was much less, it would not be as viable a technique.

Figure 28 (a-d) shows the direct comparison of overhead performance for the two FSM manipulation techniques implemented. There is approximately a 2.5% increase in overhead when going from the single insertion to double insertion fingerprinting

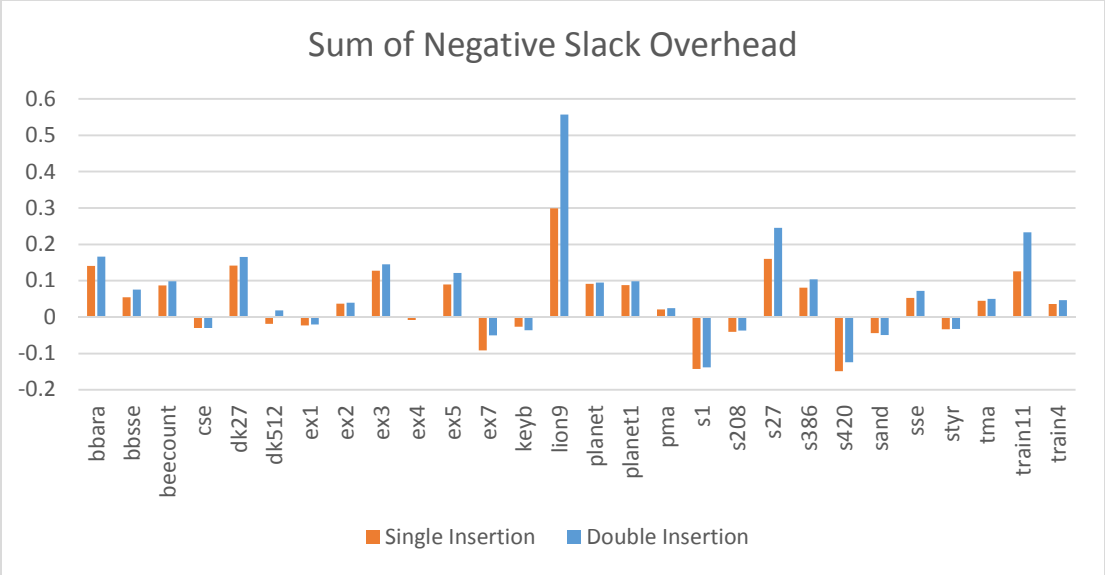
technique, and it does not vary much between the different metrics. There was a maximum difference of 3% for power and a minimum difference of 2.1% for maximum negative slack. Again, the average is high because of the size of some of the smaller benchmark circuits, namely lion9, which had an increase of approximately 18% between the two fingerprinting techniques.



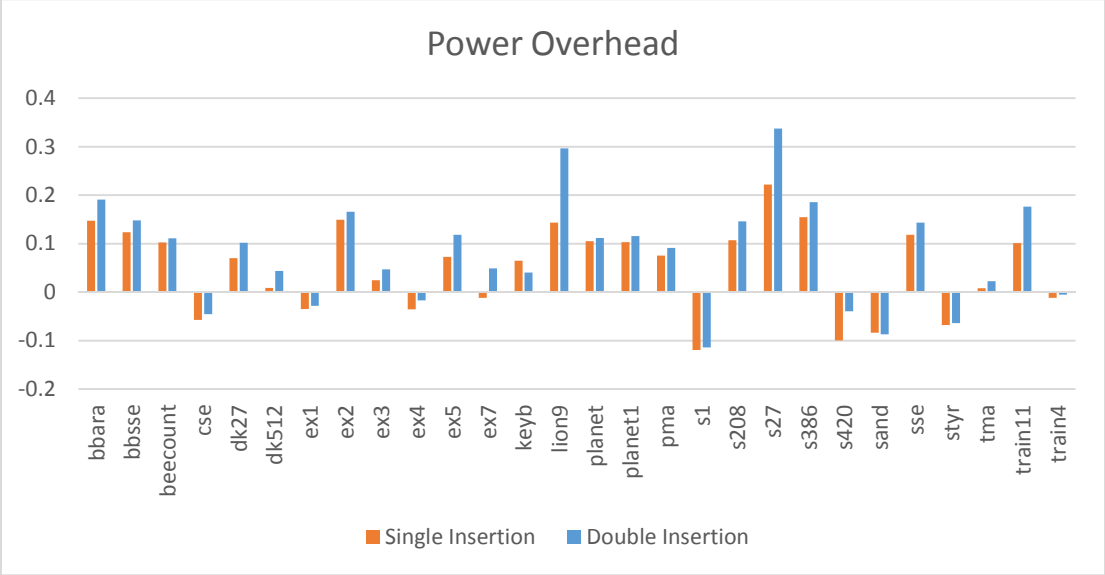
(a) Area overhead comparison



(b) Maximum negative slack overhead



(c) Sum of slack overhead



(d) Power overhead

Figure 28 (a-d). Average overhead across all benchmark circuits

The most important information for this work, and arguably every fingerprinting technique, was how many acceptable fingerprints could be generated using the technique. In this work and other fingerprinting work, the term "acceptable" is generally subjective and changes from circuit to circuit, especially with the context of

where the circuit will be used. For the sake of this experiment, "acceptable" circuits were determined to not have more than a 10% overhead for any performance metric.

With each of the experiments, as the data was collected from the sample set we tested, we checked which fingerprints met a 10% overhead threshold for every metric and divided by the sample size. The result of this is seen in Table 8. As the fingerprints were chosen randomly, this was seen as a good indicator of the number of fingerprints that could be expected to have an acceptable overhead.

Table 8. Percentage of acceptable fingerprints based on sample size

Benchmark	*% Acceptable (Single Insertion)*	*% Acceptable (Double Insertion)*
bbara	7.88%	3.93%
bbsse	4.71%	6.27%
beecount	8.65%	7.33%
cse	28.75%	27.60%
dk27	6.25%	7.81%
dk512	3.13%	1.56%
ex1	3.59%	4.22%
ex2	4.59%	3.20%
ex3	52.90%	53.67%
ex4	13.27%	11.62%
ex5	1.95%	2.80%
ex7	0.22%	0.87%
keyb	1.40%	1.60%
lion9	39.26%	12.73%
planet	3.40%	2.60%
planet1	2.79%	3.20%
pma	7.34%	6.47%
s1	1.12%	0.87%
s208	0.00%	0.00%
s27	0.78%	0.26%
s386	3.15%	2.60%
s420	0.07%	0.07%
sand	0.59%	0.80%
sse	5.77%	6.40%
styr	1.67%	1.00%
tma	12.48%	11.27%
train11	0.32%	0.33%
train4	0.00%	18.75%
Average	7.71%	7.14%

Using the data in Table 7 and Table 8, it is possible to extrapolate how many fingerprints that could be generated using the techniques in this work, and would still

be considered acceptable, as previously defined. This data can be seen in Figure 29. The chart is on a logarithmic scale so that a direct comparison between the two techniques can be easily made. This is because the number of possible fingerprints increased dramatically between the two different techniques. On average there was a 5.04e8% increase in the number of acceptable fingerprints when going from single insertion to double insertion. For that average two benchmarks were excluded, s208 which never had any acceptable fingerprints and train4 which went from zero acceptable fingerprints to 3.

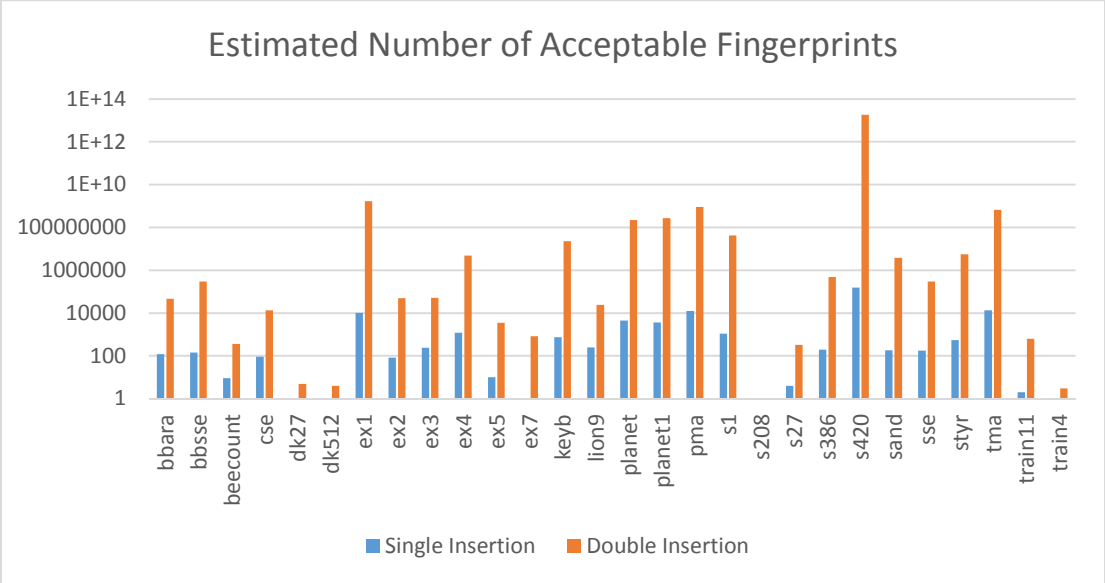


Figure 29. Number of acceptable fingerprints for both single insertion and double insertion fingerprinting techniques. Acceptable is defined as a 10% overhead maximum for every performance metric.

What this shows is that while the single insertion gave a good number of fingerprints, it may not have been enough to give unique fingerprints for each produced circuit. Fortunately, the double insertion provided enough different fingerprints with an acceptable overhead. In many cases we have an excess of potential acceptable fingerprints which can be used to specifically choose certain patterns to make the

fingerprints more secure, or to filter the acceptable fingerprints for ones that fit even more stringent threshold performance values.

E. Conclusion

In this chapter, a fingerprinting technique is proposed that utilizes the unused transitions in the FSMs for sequential circuits. This technique added one or two unused transitions to the FSM so that when mapped, the circuits could be uniquely identified. This work created a sufficient number of unique fingerprints for the circuits of a non-trivial size. These fingerprints are fairly easy to detect and are also extremely secure due to their implementation in the functional and logic design stages of the VLSI design cycle. In addition, through filtering the fingerprinted circuits, it has been shown that it is possible to get a sufficient number of fingerprints that do not cause an undue amount of overhead.

CHAPTER 6: SCAN CHAIN FINGERPRINTING

A. Scan Chain Fingerprinting

An additional solution to low-cost IP fingerprinting is the use of scan-chains in sequential systems. In this section, we will first discuss the background of scan-chains and their functionality, then we will discuss the watermarking technique proposed by [42] that we based our idea on, and finally we will discuss the method that we propose for fingerprinting ICs.

1. Background on Scan-Chain

When designing complex sequential systems, scan design can be an important step in the VLSI design cycle. Scan design attempts to add testability to an IC by adding control and observational functionality to the flip flops in a sequential system. This can allow a system to forego complex automatic test pattern generation (ATPG) [43]. Scan-chains are a technique from scan design and has been the most popular design-for-testability (DfT) technique in the EDA industry for years.

The main change that a circuit undergoes to add a scan chain, is to replace the D-Flip-Flops (DFFs) with Scan-Flip-Flops (SFFs). The SFF contains the original DFF as well as a multiplexer and two new input signals: scan-data SD and test control TC. In a scan-chain, the SFFs are chained together by connecting the output signal, Q, of one SFF to the SD input of another SFF. The TC signal is used to switch operating modes from normal to testing mode in the core under test (CUT). While in the normal operating mode, the SFFs act as the DFFs that the circuit design originally had. In testing mode test data comes in from the primary input SI, and the test results are

supplied to the primary output SO. This entire operation can be seen in Figure 30.

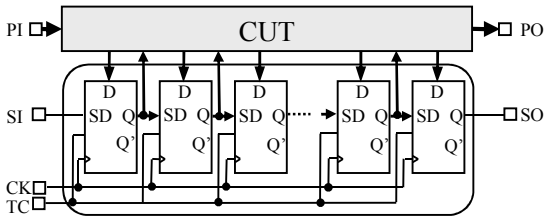


Figure 30. Scan chain design used in DfT. [42]

In most technology libraries DFFs contain two outputs Q and Q', the complement of Q. Since SFFs are built on top of DFFs, they too have the Q and Q' outputs, which can be seen in Figure 30. This secondary output allows a designer to connect Q' to the adjacent SFF SD input instead of Q. This is called the Q'-SD connection [42]. The fingerprinting method that we will propose, takes advantage of using both Q-SD and Q'-SD connection in our scan-chain.

2. Scan-chain Fingerprinting

To create a fingerprint for a sequential circuit design that uses scan-chain, we utilize the Q-SD and Q'-SD connections between SFFs. By altering which adjacent flip-flops have a Q-SD or Q'-SD connection we can create a number of patterns that would make up our fingerprint. For the n number of flip-flops that occur in the circuit, we have the potential for 2^n possible fingerprints. In addition, it would also be possible to add additional flip-flops to test parts of combinational circuitry so that we can double the number of fingerprints for each additional flip-flop.

This is an ideal solution to both our low-cost fingerprint problem while fulfilling the requirements for an effective fingerprint. With large sequential circuits, scan-chains

become almost necessary to make sure that a circuit is behaving in a properly, so the cost of changing the DFFs to SFFs as well as the additional routing and wiring is included in the design costs. Our fingerprinting method would simply require simple wiring changes so that some Q-SD connections would be switched to Q'-SD. This is significantly less costly than adding additional gates or building the circuit with new constraints. The major change that this method would cause is that the designer would need to edit the test patterns that would be put in through the scan input (SI) and reinterpret the output based on the changes that are caused by the Q'-SD connections. For example if we take a system like the one in Figure 31:

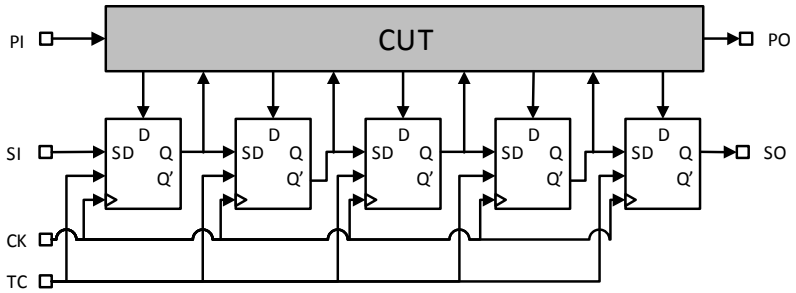


Figure 31. 5-bit Scan Chain with the second and third Q-SD connections switched to Q'-SD

If we wished to put in the bit combination 00000, representing a specific state, into the SFFs, we would need to enter in, 00110 because the first Q'-SD connection would switch them and the last one would switch the last bit back. The reverse of this would occur when trying to read a state out. If we got 00110, we would need to translate it by flipping the third and fourth bits to 00000. Note that this would not always be a symmetric process, when there are an odd number of Q'-SD connections the translation for input and output would differ.

This fingerprint scheme would be fairly easy to detect for the person who designed it. Given certain test patterns, and known responses, the designer would easily be able to detect the fingerprint. These test patterns would need to be kept secret by the designer so that adversaries would not be alerted to the presence of a fingerprint. It is also possible to do a destructive test where the designer would recognize the positions of the Q-SD and Q'-SD connections and attribute them to a certain production batch. Adversaries would only be able to detect fingerprints for the Q-SD connections that are opposite in each adversary's circuit.

In regards to removability, the result is similar to that in [42]. If an adversary wishes to remove the fingerprint they would need to reverse engineer the device, or have access to a netlist at which point they would need to remove the entire scan-chain. Reverse engineering the entire device and attempting to rebuild a new scan chain in a netlist would both be an extreme cost to the adversary, making it unlikely that they would attempt to remove or redesign the circuit without the scan chain fingerprint.

B. Security Analysis

Because this work is based on [42], the security analysis for that work mostly applies to this work as well. The major difference between this work and the work in [42] is that many more fingerprints would be created than watermarks. This means that for the coincidence security, the fingerprints should add n more bits, where n is the $\log_2$ of the number of ICs produced. This will compensate for the number of individual ICs that are going to be produced with individual fingerprints. The other security issue, false positives, is also similar to the analysis in [42]. The only change would

be that the random test vectors that are applied would need to be adjusted for the different fingerprint configurations.

The other major attack that adversaries could carry out is either forging a fingerprint or changing the test vectors associated with a certain device, making it is difficult or impossible to identify the fingerprint. Both of these attacks are difficult to perpetrate either because they involve redesigning the circuit or because adjusting the test vectors will lead to a lower test coverage of the device. Without the proper test coverage, a circuit may be malfunctioning and the end user may not know.

C. Experimental Setup

In the experiment, we used the Design Compiler under Synopsys to synthesize and obtain netlists from the designs from ISCAS89, ISCAS99 and LGSynth93 benchmark suites. The DfT Compiler and TetraMax under Synopsys are, respectively, used to create the original scan chain and generate the test patterns. The 64-bit, 128-bit and 256-bit watermarks are, respectively, embedded into the experimental designs. All experiments were run on a 3GHz HP Z620 work station with Linux operating system and 12 GB of memory.

D. Results

Table 9 shows the fingerprinting results on the ISCAS and LGSynth93 benchmark circuits using the proposed fingerprinting method. The columns of 'N' and 'T_{org}' denote the length of the scan chain and the number of transitions during testing by the originally optimized scan design, respectively. ΔT represents the percentage increments from T_{fp} to T_{org}, where T_{fp} denotes the number of transitions during testing by the

fingerprinted scan design. The column, '*n*' under that of '**Δ*T***', denotes the maximum number of connections among (*N*-1) connections that can be altered by fingerprinting while maintaining the overhead on transitions smaller than **Δ*T***. To evaluate the overhead due to multiple different fingerprints, we use the pseudo-random generator (PNG) to generate 10 random numbers between [1..*n*] to index 10 connections among the *n* qualified connections. We then compute the average overhead of transitions caused by the 1024 different fingerprinted designs, which are implemented by different configuration of the 10 selected connections. The column '**Δ*AT***' denotes the average overhead of transitions from the 1024 fingerprinted designs. We can see that for a design, a smaller **Δ*T* corresponds to a smaller *n*, which means a smaller pool of the qualified connections. To guarantee the overhead less than 0.1%, at least 70 qualified connections (the design S35932) can be found. This can enable a sufficiently large pool of 2^{70} fingerprints. Also, the percentage average overhead of 1024 different fingerprinted design can be controlled not more than 7.00E-03.**

Table 9. Average Overheads of Transitions due to 1024 different fingerprints on Benchmark Circuits

Circuit	*N*	T_{org}	Δ*T*= 1%		Δ*T*=0.5%		Δ*T*=0.2%		Δ*T*=0.1%	
			n	Δ*AT*(%)	*N*	Δ*AT*(%)	*N*	ΔAT(%)	*n*	Δ*AT*(%)
S38584	1166	3.31E+08	458	1.11E-02	329	7.07E-03	207	5.46E-03	148	4.00E-03
S38417	1564	1.13E+09	599	9.06E-03	426	7.96E-03	276	3.22E-03	199	2.21E-03
S35932	1728	6.72E+07	231	1.41E-02	161	1.38E-02	101	1.26E-02	70	8.76E-03
B17	1315	2.31E+09	366	1.69E-02	266	1.11E-02	168	7.48E-03	120	3.16E-03
B17_1	1316	2.31E+09	383	1.23E-02	285	5.46E-03	185	4.35E-03	135	3.91E-03
B18	2908	2.45E+10	1059	4.70E-03	779	3.69E-03	500	1.81E-03	355	1.16E-03
B18_1	2904	2.30E+10	1069	3.21E-03	782	3.26E-02	500	1.81E-03	353	1.40E-03
B19	5816	1.55E+11	2306	1.88E-03	1770	1.30E-03	1156	7.39E-04	826	7.30E-04
B19_1	5709	1.44E+11	2263	2.57E-03	1735	1.39E-03	1133	6.85E-04	803	6.88E-04
DMA	1831	2.21E+09	**793**	**5.03E-03**	572	5.19E-03	373	2.08E-03	269	1.65E-03
usb_funct	1517	1.30E+09	**456**	**8.53E-03**	326	6.89E-03	205	4.65E-03	142	2.93E-03
ac97_ctrl	1876	6.73E+08	**599**	**9.13E-03**	426	7.73E-03	268	3.96E-03	188	2.71E-03
pci_bridge32_1	1485	9.30E+08	**661**	**9.40E-03**	474	5.72E-03	301	3.57E-03	212	2.70E-03
pci_bridge32_2	1828	1.40E+09	**786**	**6.51E-03**	563	4.26E-03	359	2.19E-03	255	2.04E-03
des_perf	8808	1.32E+10	**3575**	**1.29E-03**	2550	7.91E-04	1622	6.79E-04	1154	4.49E-04
ethernet	10015	1.37E+11	**2042**	**2.76E-03**	1609	1.45E-03	1079	1.08E-03	771	6.95E-04
vga_lcd	16904	1.59E+12	**7056**	**5.30E-04**	5978	4.16E-04	3993	3.21E-04	2862	1.85E-04
Average	--	--	--	7.00E-03	--	**6.87E-03**	--	**3.33E-03**	--	**2.32E-03**

E. Conclusion

Scan chain fingerprinting is an ideal solution to fingerprinting circuits that utilize scan-chains for DfT. The overhead is minimal as its only real affect is to increase the power usage of the device during testing. This method can also create fingerprints with more than sufficient length for most production lines. With this we can create larger than necessary fingerprints that can either be entangled or simply include more information, making it more difficult for attackers to counterfeit or break.

CHAPTER 7: FUTURE WORK

A. FSM Manipulation

There are a number of improvements that could be made to our third method, the FSM manipulation fingerprinting technique. The work presented in this dissertation was limited to single and double transition insertions, but could easily be expanded include more complicated FSM manipulations. First, the number of transitions that are inserted could be increased. This would need to be analyzed further because there may be a tipping point where adding more transitions will cause unacceptable overhead for all possible fingerprints. It may also be possible to determine the transitions that cause the least overhead and focus on them more to create faster processes for generating FSM fingerprints.

Another future improvement to this work includes utilizing transitions outputs to create more fingerprints. The work in this dissertation simply focused on adding in specific transitions based on an input, starting state, and ending state, but the output was left as a *don't care* so that the mapping software could potentially make the circuit more efficient. This cut down on the number of potential fingerprints, and it is possible that like the states that were added, there may be some outputs that if specified will actually be an improvement over the results we measured with the *don't care* output.

The final improvement that is being considered, is adding new states or inputs to increase the number of unused edges. All of the benchmarks that were tested were missing transitions so this was unnecessary, but in situations where there are very few

missing transitions, or the FSM is fully specified, there may not be enough fingerprints to uniquely identify each manufactured circuit. In this situation, additional inputs or flip flops may be added to the device to significantly increase the number of unused transitions, and only require a small increase in the performance overhead.

B. Scan-chain

There are also plans to improve the scan-chain work by testing the security of test vectors. It is possible to test the test coverage change due to an adversary's manipulation of the test vectors. From here, it can be seen what kind of IC manufacturing errors an adversary would miss and how it could potentially affect counterfeit circuits.

CHAPTER 8: CONCLUSION

IP theft, duplication, and modifications are a large potential problem in the IC market. It is important that methods are developed to mitigate this issue, without creating undue cost to designers and fabricators. Fingerprinting is one such method that allows IP to be tracked and accounted for. Fingerprints allow designers to individually mark their ICs such that they can be tracked from manufacturing to end-user, if they are registered. They allow an end-user to make sure that they have a legitimate copy of the device and one that was not resold as new.

This dissertation has discussed four methods for creating lower cost fingerprinting techniques for ICs. Our first method took advantage of observability don't cares that appear in virtually every IC of significant size. We also developed a similar method that takes advantage of satisfiability don't cares, for even less intrusive fingerprints for a circuit. Our third method uses FSM modifications to change the ICs structure without affecting its core functionality. The fourth and final method uses modifications to the Scan-chain interconnects to create fingerprints that are easy to detect and have extremely small overhead.

Each of these methods can automatically be implemented into a design with a minimal performance overhead. The ODC, SDC, and Scan-chain methods were developed so that by injecting flexibilities into the design by the way of fuses or wires that can be cut, a developer has the option of adding in a fingerprint in the post-silicon stage, thus reducing the manufacturing costs associated with older fingerprinting methods.

REFERENCES

[1] A. Cui and C.-H. Chang, "Stego-signature at Logic Synthesis Level for Digital Design IP Protection," in *ISCAS*, Island of Kos, 2006.

[2] I. Hong and M. Potkonjak, "Techniques for intellectual property protection of DSP designs," in *IEEE International Conference. Acoustics, Speech, and Signal Processing*, Munich, Germany, 1998.

[3] A. E. Caldwell, H.-J. Choi, A. B. Kahng, S. Mantik, M. Potkonjak, G. Qu and J. L. Wong, "Effective Iterative Techniques for Fingerprinting Design IP," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, New York, NY, 1999.

[4] A. B. Kahng, S. Mantik, I. L. Markov, M. Potkonjak, P. Tucker, H. Wang and G. Wolfe, "Robust IP watermarking methodologies for physical design," in *Proceedings of the ACM/IEEE Design Automation Conference*, Piscataway, NJ, 1998.

[5] U. Guin, K. Huang, D. DiMase, J. M. Carulli, Jr., M. Tehranipoor and Y. Makris, "Counterfeit Integrated Circuits: A Rising Threat in the Global Semiconductor Supply Chain," *Proceedings of the IEEE,* vol. 102, no. 8, pp. 1207-1228, 2014.

[6] IHS Technology, "Reports of Counterfeit Parts Quadruple Since 2009, Challenging US Defense Industry and National Security," 14 February 2012.

[Online]. Available: https://technology.ihs.com/389481/. [Accessed 24 March 2015].

[7] Aerospace Industries Association, "Counterfeit Parts: Increasing Awareness and Developing Countermeasures," AIA, Arlington, VA, 2011.

[8] N. A. Sherwani, Algorithms For VLSI Physical Design Automation Third Edition, New York: Kluwer Academic Publishers, 2002.

[9] V. M. Potdar, H. Song and E. Chang, "A suvey of digital image watermarking techniques," in *IEEE International Conference on Industrial Informatics*, 2005.

[10] E. Charbon and I. H. Torunoglu, "On Intellectual Property Protection," in *Custom Integrated Circuits Conference*, Orlando, FL, 2000.

[11] R. G. Schyndel, A. Tirkel and C. F. Osborne, "A Digital Watermark," in *Proceedings of IEEE International conference on Image Processing*, 1994.

[12] S. Prabhishek and R. S. Chadha, "A Survey of Digital Watermarking Techniques, Applications, and Attacks," *International Journal of Engineering and innovative Technology,* vol. 2, no. 9, pp. 165-173, 2013.

[13] N. R. Wagner, "Fingerprinting," in *Proc. of the 1983 IEEE Symposium on Security and Privacy*, Washington D.C., 1983.

[14] G. Caronni, "Assuring ownership rights for digital images," in *Verläßliche IT-Systeme*, Germany, Vieweg+Teubner Verlag, 1995, pp. 251-263.

[15] I. Cox, J. Kilian, T. Leighton and T. Shamoon, "A secure robust watermark for multimedia," in *Information Hiding*, New York, Springer-Verlag, 1996, pp. 185-206.

[16] "DigiMarc Co.," [Online]. Available: http://www.digitmark.com.

[17] K. Tanaka, Y. Nakamura and K. Matsui, "Embedding secret information into a dithered multi-level image," in *Proc. 1990 IEEE Military Communications Conf.*, Monterey, CA, 1990.

[18] C.-Y. Lin, M. Wu, J. A. Bloom, I. J. Cox, M. L. Miller and Y. M. Lui, "Rotation, Scale, and Translation Resilient Watermarking for Images," *IEEE Transactions on Image Processing,* vol. 10, no. 5, pp. 767-782, 2001.

[19] D. Boneh and J. Shaw, "Collusion secure fingerprinting for digital data," *IEEE Transactions on Information Theory,* vol. 44, no. 5, pp. 1897-1905, 1998.

[20] D. Glover, The Protection of Computer Software, 2nd ed., Cambridge, UK: Cambridge Univ. Press, 1992.

[21] E. Charbon, "Hierarchical watermarking in IC design," in *Proc. Custome Integrated Circuit Conf.*, Santa Clara, CA, 1998.

[22] I. Torunoglu and E. Charbon, "Watermarking-Based Copyright Protection of Sequential Functions," *IEEE Journal of Solid-State Circuits,* vol. 35, no. 3, pp. 434-440, 2000.

[23] A. L. Oliveira, "Techniques for the creation of digital watermarks in sequential circuit designs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* vol. 20, no. 9, pp. 1101-1117, 2001.

[24] H. J. Patel, J. W. Crouch, Y. C. Kim and T. C. Kim, "Creating a unique digital fingerprint using existing combinational logic," in *Circuits and Systems, IEEE International Symposium on*, Taipei, 2009.

[25] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi and B. Sunar, "Trojan Detection Using IC Fingerprinting," in *IEEE Symp. Security and Privacy*, Berkeley, CA, 2007.

[26] Y. Jin and Y. Makris, "Hardware Trojan detection using path delay fingerpring," in *Hardware-Oriented Security and Trust, IEEE International Workshop on*, Anaheim, CA, 2008.

[27] J. Lach, W. H. Mangione-Smith and M. Potkonjak, "Fingerprinting techniques for field-programmable gate array intellectual property protection," *Computer-Aided Design of Integrated Circuits and Systems,* vol. 20, no. 10, pp. 1253-1261, 2001.

[28] J. Lach, W. Mangione-Smith and M. Potkonjak, "Fingerprinting Digital Circuits on Programmable Hardware," in *Information Hiding*, Berlin Heidelberg, Springer-Verlag, 1998, pp. 16-31.

[29] K. W. Yip and T. S. Ng, "Partial-Encryption Technique for Intellectual Property Protection of FPGA-Based Products," *IEEE Transactions on Consumer Electronics,* vol. 46, no. 1, pp. 183-190, 2000.

[30] A. K. Jain, L. Yuan, P. R. Pushkin and G. Qu, "Zero overhead watermarking technique for FPGA designs," in *Proceedings of the 13th ACM Great Lakes Symposium on VLSI*, 2003.

[31] A. B. Kahng, D. Kirovski, S. Mantik, M. Potkonjak and J. L. Wong, "Copy Detection for Intellectual Property Protection of VLSI Desgin," in *IEEE/ACM*

International Conference on Computer-Aided Design, San Jose, California, 1999.

[32] S. Megerian, M. Drinic and M. Potkonjak, "Watermarking Integer Linear Programming Solutions," in *IEEE/ACM Design Automation Conference*, New Orleans, LA, 2002.

[33] G. Qu and M. Potkonjak, "Fingerprinting intellectual property using constraint-addition," in *Proceedings of the 37th Annual Design Automationn Conference*, New York, NY, 2000.

[34] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Sal-danha, H. Savoj, P. R. Sephan, R. K. Brayton and A. Sangiovannie-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," University of California, 1992.

[35] R. K. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *Computer Aided Verification*, Berlin Heidelberg, Springer, 2010, pp. 24-40.

[36] K. Chang, I. L. Markov and V. Bertacco, "Automating Post-Silicon Debugging and Repair," in *IEEE/ACM Intl Conference on Computer Aided Design*, San Jose, CA, 2007.

[37] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translation in FORTRAN," in *Proc. IEEE Int. Symp. Circuits Syst.*, 1985.

[38] F. Brglez, D. Bryan and K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," in *IEEE Intl Symp. on Circuits and Systems*, Portland, OR, 1989.

[39] S. Yang, "Logic Synthesis and Optimization Benchmarks," Microelectronics Centre of North Carolina, 1991.

[40] "Open Source Digital Flow," 26 August 2013. [Online]. Available: http://opencircuitdesign.com/verilog/. [Accessed 14 July 2014].

[41] E. Times-Asia/Gartner, "Design Trends and EDA Tools: China & Taiwan," EMEDIA Asia LTD. and Gartner, Inc., 2005.

[42] A. Cui and G. Qu, "Dynamic Watermarking on Scan Design for Hard IP Protection with Ultra-low Overhead," 2014.

[43] M. L. Bushnell and V. D. Agrawal, Essentials of Electronic Testing, Kluwer Academic Publishers, 2000.

[44] A. T. Abdel-Hamid, S. Tahar and E. M. Aboulhamid, "IP Watermarking Techniques: Survey and Comparison," in *System-on-Chip for Real-Time Applications*, 2003.

[45] L. Yuan, G. Qu, T. Villa and A. Sangiovanni-Vincentelli, "FSM Re-Engineering: A Novel Approach to Sequential Circuit Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, no. 7, pp. 1230-1246, 2006.

Designing Trusted Embedded Systems from Finite State Machines

CARSON DUNBAR, Department of Electrical and Computer Engineering, University of Maryland, College Park.
GANG QU, Department of Electrical and Computer Engineering, University of Maryland, College Park.

***Abstract*—** Sequential components are crucial for real time embedded systems as they control the system based on the system's current state and real life input. In this paper, we explore the security and trust issues of sequential system design from the perspective of finite state machine (FSM), which is the most popular model to describe sequential systems. Specifically, we find that the traditional FSM synthesis procedure will introduce security risks and cannot guarantee any trustworthiness in the implemented circuits. Indeed, we show that not only there exist simple and effective ways to attack a sequential system, it is also possible to insert hardware Trojan Horse into the design without introducing any significant design overhead. We then formally define the notion of trust in FSM and propose a novel approach to designing trusted circuits from the FSM specification. We demonstrate both our findings on the security threats and the effectiveness of our proposed method on MCNC sequential circuit benchmarks.

1. Introduction

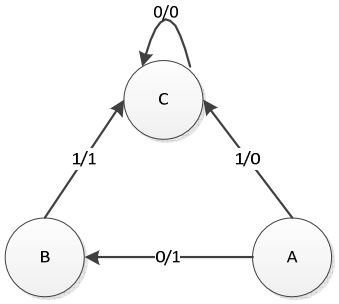
As electronic design automation (EDA) and semiconductors continue to evolve rapidly, a company will not have all the expertise and capability to do in-house design and fabricate the system it wants to build. If the system is designed and fabricated by others, how can the company be convinced that the system is trusted, that is, the delivered system does exactly what the company wants, no more and no less. This is known as the trusted integrated circuits (IC) design challenge, particular for military and civilian systems that require security and access control [1-8].

When the company gives the system's specification to a design house for layout and then gives the layout information to a foundry for manufacture, the company will lose full control of the system's functionality and specification. An adversary can simply add additional circuitry, known as a hardware Trojan horse, to maliciously modify the system. For example, a Trojan horse can: disable or destroy system components, perform incorrect computation, or leak sensitive information. Most of the existing work on trusted IC design focuses on hardware Trojan detection and prevention [9-13].

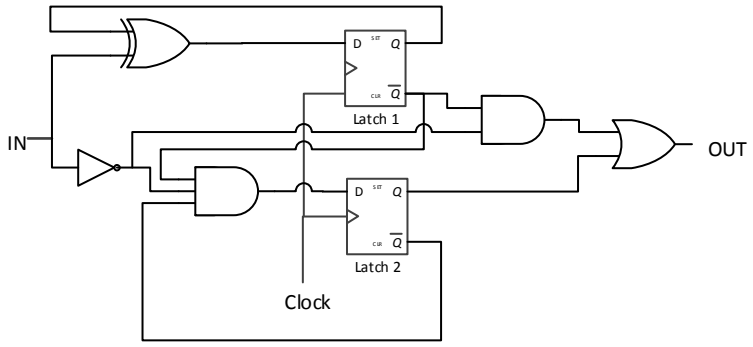
In this paper, we explore the vulnerabilities of sequential systems designed by today's design methodology. More specifically, we consider the following questions:

1. When a sequential system is designed and implemented by a trusted party strictly following the design specification, can we trust it?
2. When an adversary inserts hardware Trojan into the system, can we detect it?
3. What is the design cost to build an ideal trusted IC, if it exists?

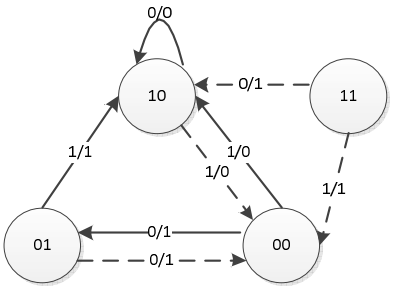
We now elaborate these questions by the following illustrative example. Consider the 3-state finite state machine (FSM) shown in Figure 1(a); we follow the standard procedure to implement this sequential system with two flip flops (FF) and some logic gates (Figure 1 (b)). First, from Figure 1(a), we see that when the system is at state B and input is 0, no next state and output are specified, which are known as *don't care transitions*. However, in the circuit level implementation when FF1 is 0 and FF2 is 1, which reflects the current state B, if input $x = 0$, we can easily verify that FF1 remains unchanged, but FF2 changes to 0. This means that the system switches to state A. At the same time, we can see that the output will be 1. This corresponds to the dashed line from state 01 to state 00 in Figure 1(c). Similarly, state 10 will move to state 00 and output 0 on input $x = 1$.



(a) The original 3-state FSM as the system specification. The label on each edge (such as 0/1 from state A to state B) indicates that the transition occurs on input '0' and the transition results in an output '1'.



(b) The logic/circuit implementation of the 3-state FSM shown in (a).



(c) The 4-state FSM generated from the circuit shown in (b).

Figure 1: The illustrative example. States A, B, and C in (a) correspond to the states 00, 01, and 10, respectively in (c). The dashed edges are the transitions implemented by the circuit in (b) but not required by the FSM in (a).

Second, when both flip flops have value 1, the system will be in a state that is not specified in the original FSM. With input $x = 0$ and $x = 1$, the system will output 1 and move to state C and state A, respectively. In other words, the full functionality of the circuit in Figure 1(b) can be described as is shown in Figure 1(c).

The FSM in Figure 1(a) is what we want to design, the FSM in Figure 1(c) is the system that the circuit we are given (Figure 1(b)) actually implements. Clearly we can see the difference between these two FSMs. The one in Figure 1(c) has one more state and four more transitions. It is this added state and transitions that creates security and trust concerns for sequential system design.

In the original FSM (Figure 1(a)), when the system leaves state A, it cannot come back. In other words, we say that there is no access to state A from state B and state C. However, in the implemented FSM in Figure 1(c), each state has at least one path to come back to state 00. For instance, from state

01 (which is state B in the original FSM), when we inject 0 as the input, the system moves back to 00 (or state A). If state A is a critical state of the system and we want to control its access, FSM in Figure 1(a) specifies this requirement, but its logic implementation in Figure 1(b) violates this access control to state A (or state 00) and the design cannot be trusted.

This example shows that there are security holes in current sequential system design flow. In this paper, we study how an attacker can take advantage of these holes to attack the system. We analyze the cost to detect and prevent such attacks. Then we propose a novel method that can be seamlessly integrated into the current sequential system design flow to establish trust in the design and implementation. We conclude the introduction with a brief survey of the related work on trusted IC design, hardware Trojan, and FSM watermarking.

The challenge of building trust in IC is highlighted in a Defense Science Board study on High Performance Microchip Supply, "*Trust cannot be added to integrated circuits after fabrication; electrical testing and reverse engineering cannot be relied on to detect undesired alternations in military integrated circuits*" [1]. The Trusted Foundry Program and the complementary Trusted IC Supplier Accreditation were specifically designed for domestic fabrication, where trust and accreditation are built on reputation and partnership [2]. The notions of trusted and trustworthiness are presented in [3], where the authors discuss the challenges, opportunities, call for new initiatives and programs to establish principles, tools, and standards for building trusted hardware. Trimberger in [4] argues that trusted IC can be built on a field programmable gate array (FPGA) platform because it separates the manufacturing process from the design process. However, the base array still needs to be verified through manufacture and trust still needs to be built during the design process. Suh and Devadas [5] propose physical unclonable functions (PUFs) based on transistor and wire delay to uniquely identify a chip. Logic obfuscation techniques have been proposed recently to solve the IC piracy problem and build trusted IC [6, 7]. However, none of these efforts can be used to verify whether the chip contains unwanted functionalities. Gu et al. [8] develop information hiding method for trusted system design where they impose additional design constraints in the hope that design with unwanted functionalities will incur noticeable performance degradation and thus can be caught. This novelty concept is hard to be implemented due to the complexity of design process.

Hardware Trojan refers to any kind of malicious modification of the IC. It is one of the more serious threats to trusted IC design. Banga and Hsiao [9] propose a circuit partition based approach to detect and locate the embedded Trojan. Rad et al. [10] develop a power supply transient signal analysis method for detecting Trojans based on the analysis of multiple

power port signals to determine the smallest detectable Trojan. Wang et al. [11] explore the wide range of malicious alternations of ICs that are possible and propose a general framework for their classification as well as several Trojan detection strategies. Gong and Makkes [12] present a Trojan side-channel based on PUF that can successfully attack block ciphers. Wei et al. [13] develop a set of hardware Trojan benchmarks that are the most challenging representative test cases for side-channel based hardware Trojan detection techniques. The existing hardware Trojan detection approaches rely on catching the misbehavior of the IC caused by the hardware Trojan embedded into a known design. When a hardware Trojan is added during the design process and integrated with the required functionalities of the IC, these approaches will not be effective.

There is a rich body of research work on FSM watermarking, for the protection of FSM design intellectual property [14-19]. These techniques usually rely on the modification of the state transition graph at the behavioral synthesis level to embed watermark related to user-specific information for identification purpose. In [14], Oliveira proposes to create watermarks based on a set of redundant states which can only be traversed when a user-specific input sequence is loaded. Lewandowski et al. [15] and Zhang and Chang [16] propose watermarking schemes based on state encoding. Torunoglu and Charbon [17] introduce extra state transitions in the FSM to produce output that carries watermark. Abdel-Hamid et al. [18] improve this method by utilizing the existing transitions for watermarking and successfully reduce the high-overhead caused by extra state transitions. Cui et al. [19] propose an improved scheme that increases the ratio of the number of existing transitions used during the watermarking process to further reduce overhead. The concept behind these FSM watermarking techniques is to embed additional information into the FSM, which is similar to hardware Trojan insertion. However, the added information is for authorship proof and normally does not carry any malicious functionality (like a hardware Trojan does).

The rest of the article is organized as follows: in Section 2, we give the necessary background of finite state machine. In Section 3, we elaborate the unsecured safe-state vulnerability in the current design, introduce two simple but powerful attacks, and propose countermeasures. We conduct experiments on standard benchmark circuits and report the results in Section 4. Section 5 concludes the article.

2. Background of Finite State Machine

A finite state machine (FSM) is defined as a 6-tuple $<I, S, \delta, S0, O, \lambda>$ where:

I is the input alphabet;

S is the set of states;

δ: $S \times I \rightarrow S$ is the next-state function;

$S0 \subseteq S$ is the set of initial states;

O is the output alphabet;

λ : $S \times I \rightarrow O$ is the output function.

An FSM can be conveniently represented as a directed weighted (or labeled) graph $G = (V, E)$, where each vertex $v \in V$ represents a state $s \in S$; an edge $(u, v) \in E$ represents the transition from current state u to its next state v, the weight (or label) on the edge indicates the input-output pair determined by the output function λ. That is, if the edge (u, v) is labeled x/y, then we have $\delta(u, x) = v$ and $\lambda(u, x) = y$. (See Figure 1(a) and Figure 1(c) for an example). Such graph is often referred to as *state transition graph*.

An FSM is *completely specified* if both the next-state function δ and the output function λ are defined on all possible current state and input pairs (u, x). Otherwise, either δ, λ or both will be undefined on some current state and input pairs (u, x). When δ is undefined, we call this a *don't care transition*; when λ is undefined, its output function has a *don't care*. The FSM is called *incompletely specified* if this happens.

We say that a state v is reachable from state u if and only if there is a directed path from u to v, that is, there is an input sequence following which the state will move from u to v. We define the *reachable set of state* u as

$$R(u) = \{v \in V \mid v \; is \; reachable \; from \; u\}$$

which is the set of all states that the system can reach from u and the *starting set of state* u as

$$S(u) = \{v \in V \mid u \; is \; reachable \; from \; v\}$$

which is the set of states from which the system can reach u.

For example, in Figure 1(a), $R(A) = \{B, C\}$, $R(B) = \{C\}$, $R(C) = \{C\}$, $S(A) = \phi$, $S(B) = \{A\}$, and $S(C) = \{A, B\}$, where $S(A) = \phi$ because there is no state transition going to state A.

Considering the FSM M' generated from the circuit implementation of a given FSM M, based on whether we want to control the access to a state in M and the reachability of the state in M', the states in M can be partitioned into three groups:

- A state v is *safe* if we want to control the access to v in M and v can only be accessed in M' from its starting states $S(v)$.
- A state v is *unsafe* if we want to control the access to v in M but v can be accessed in M' from states that are not in $S(v)$.
- A state v is *normal* if we do not want to control the access to v in M.

An adversary may attempt to gain access to a protected state v from states that are not in $S(v)$. If the adversary succeeds, the state becomes unsafe. Otherwise, all the states that we want to protect are safe and the implementation of the FSM is trusted. The goal of trusted sequential system design is to guarantee that the circuit implementation will be trusted.

We conclude this section with the two standard phases of FSM synthesis: state minimization and state encoding, both of which are critical in our proposed method to establish trust in FSM implementation.

Two FSMs are *equivalent* if from the initial state, on any input sequence, they both create the same output sequence. Finding an equivalent FSM with minimal number of states is generally referred as *state minimization* or *state reduction* problem. State minimization is an effective approach in logic synthesis to optimize sequential circuit design in terms of area and power. It can be solved optimally in completely specified FSMs and the solution is unique [20]. For incompletely specified machines, although the problem is NP-hard, there are standard approaches to solve the state reduction problem [21].

The next phase of FSM synthesis is *state assignment* or *state encoding* where the goal is to assign distinct binary codes to each state of the FSM such that the sequential circuit modeled by the FSM can be efficient in terms of area, performance and/or power. There have been many techniques to solve the state encoding problem based on different optimization objectives and implementation technologies [22]. Each bit of the binary code will be implemented by one flip flop and the combinational part of the circuitry can be designed to generate input signals to each flip flop and produce the desired output. This will give us a logic implementation of the FSM and concludes the sequential system design.

3. Trust, Attacks, and Countermeasure

3.1 Trusted FSM and Trusted Logic Implementation

Intuitively, we can define trust as follows: when a sequential system is specified as an FSM, it is trusted as long as the FSM makes correct transitions from the current state to the next state and produces correct outputs based on the input values. From this definition, it is clear that if an FSM is trusted, all of its equivalent FSMs will be trusted, which implies that whether an FSM is trusted will not change during the state minimization phase. However, this may change during the state encoding and combinational logic design phase of the FSM synthesis.

First, as we have seen from the illustrative example in Figure 1, additional states and additional transitions may be introduced when we design the logic. In the state transition graph, we can have *don't care* conditions where

the next state or the output of the transition or both are not specified. Logic design tools will take advantage of these *don't care* conditions to optimize the design for performance improvement, area reduction, or power efficiency. But when the system (or the FSM) is implemented in the circuit level, these *don't cares* will disappear. The circuit will generate deterministic next states and output for each of the *don't care* conditions. These deterministic values are assigned by CAD tools for optimization purpose and they may make the design untrusted. For example, the next state of a *don't care* transition may be assigned to a state for which access control is required and thus produce an illegal entry to that protected state (i.e., making the state unsafe).

A second type of implicit violation of trust comes from the nature of digital logic implementation. When the original FSM has n states after state minimization, it will need a minimum of $k = \lceil log_2(n) \rceil$ bits to encode these states and some encoding schemes that target other design objectives (such as testability and power) may use even longer codes. As we have seen in the illustrative example, when n is not a power of 2, which happens most of the time, those unused codes will introduce extra states into the system, and all transitions from those extra states will be treated as *don't care* transitions during logic synthesis, introducing uncertainty about the trust of the design and implementation of the FSM.

By analyzing the logic implementation of a given FSM M, we can build an FSM M' that captures the behavior of the circuit. When M and M' are equivalent, we say that the logic implementation is trusted. From the above discussion, we conclude that

Theorem 1. A sequential system will have a trusted logic implementation from the traditional synthesis flow if and only if

a) the system is completely specified
b) the number of states after state reduction is a power of 2
c) code with the minimal length is used for state encoding.

[Proof]: The analysis above shows all the three conditions are necessary. To see they are also sufficient, condition a) indicates that there is no *don't care* transitions; conditions b) and c) guarantee that in the implementation of the system there is no additional states. Therefore, the state transition graph of the sequential system will be unique and cannot be modified. This ensures that the system will be trusted. ■

An ideal trusted IC can be built if the system satisfies the three conditions in Theorem 1. However, it is unrealistic to assume that conditions a) and b) will be satisfied. First, given the complexity of today's system, it is impossible to completely specify the system's behavior of all possible input values. Second, there are no effective methods to ensure that the number of state, after state minimization, will be a power of 2. Finally, without any

don't cares (condition a)) and no flexibility in choosing the code length (condition c)), the design will be tightly constrained and hard to optimize. In the experimentation, when we modify the system specification to comply with conditions a)-c), the quality of design drops significantly in terms of area, power, and delay. Detailed experimental results can be found in Section 4.

In the rest of this article, we study the trust of FSMs using state reachability as a metric. Within this context, we consider a given FSM, $M = (V, E)$ (e.g. Figure 1(a)), and its logic implementation (e.g. Figure 1(b)), let $M' = (V', E')$ be the completely specified FSM generated from the logic implementation of M (e.g. Figure 1(c)). Clearly, as graphs, M will be a subgraph of M'. We say that ***the logic implementation of M is trusted*** if for each state $v \in V$ and its corresponding state $v' \in V'$, v and v' have the same reachable sets $R(v) = R(v')$ and the same starting sets $S(v) = S(v')$. Intuitively, this means that in M', we cannot reach any new states from v ($R(v) = R(v')$) and no new state can reach v either ($S(v) = S(v')$). Apparently, the logic implementation in Figure 1(b) and the corresponding FSM in Figure 1(c) cannot be trusted.

Theorem 2. The following are equivalent definitions for trusted logic implementation: for any state $v \in V$ in an FSM M and its corresponding state $v' \in V'$ in the logic implementation of M,

(1) $R(v) = R(v')$ *and* $S(v) = S(v')$
(2) $R(v) = R(v')$
(3) $S(v) = S(v')$

[Proof]: We need to show that (1) ⇔ (2) ⇔ (3). Since (1) is the conjunction of (2) and (3), it suffices to show that (2) ⇔ (3). We prove (2) ⇒ (3) by contradiction as follows. (3) ⇒ (2) can be proved similarly.

If (2) holds, but (3) does not, then there must exist a pair of states $v \in V$ and its corresponding state $v' \in V'$ such that $R(v) = R(v')$ but $S(v) \neq S(v')$. That is, we can find a state $u \in S(v)$ but its corresponding state $u' \notin S(v')$ or vice versa. From the definition, we know that $u \in S(v)$ is equivalent to $v \in R(u)$. Hence, if we have $u \in S(v)$ but $u' \notin S(v')$ as we just found, we should also have $v \in R(u)$ but $v' \notin R(u')$, which implies that $R(u) \neq R(u')$, contradicting the assumption that (2) holds. ■

3.2 Attacks to untrusted FSMs

We consider the following two attacking scenarios for the sequential system based on what the adversary can access:

Case I: the adversary can only access the logic implementation of the system or FSM M'. The attacking objective is to gain access to the states that are not accessible as specified in the original specification M. That is, finding paths in M' to states that are unreachable in M.

Case II: the adversary gets hold of the original system specification, M, in the format of FSM and wants to establish a path to reach certain unreachable state without being detected. In this case, the attacker can implement such path into the design. However, the challenge is how to disguise the secret path.

We describe two naive attacks, one for each case. As we will show in the experimental results section, these two simple attacks turn out to be quite powerful and challenging to defend. Therefore, we do not consider any sophisticated attacks although they can be developed.

Attack I: The adversary is aware of the vulnerability of the logic implementation of the FSM following the traditional design flow. Therefore, he can launch the "***random walk attack***" and hope to gain access to states that he is not supposed to reach. In this attack, the adversary will try random input sequences. If it leads to the discovery of a previously safe state (i.e., states that cannot be reached by the adversary according to the design specification), the attack will be successful. This is possible because the FSM synthesis tools will assign values to the *don't care* transitions in order to optimize design objectives such as delay, area, or power. These added transitions may make some of the safe states reachable from states that do not belong to their starting states set and therefore, making them unsafe.

Attack II: In this case, the adversary has the original FSM specification of the system before it is synthesized. If he wants to access state v from a state $u \notin S(v)$, the adversary can simply do the following:

- Check whether there is any *don't care* transition from u, if so, he simply makes v as the next state for that transition. This will give him an unauthorized access to state v in the logic implementation of the system.
- If the state transitions from state u are all specified, he can check whether there are any *don't care* transitions from a vertex/state that belongs to $R(u)$, and try to connect that state to v to create a path from u to v.
- If this also fails, then state v in the system is safe with respect to state u in the sense that one can never reach state v from state u. In this case, the attack fails.

Finally, we mention that in case II, the adversary can take advantage of the new states that logic synthesis tools will introduce (that is, when the number of states is not a power of 2 or non-minimal length encoding is used). He can simply launch attack by connecting any of the new states to state v to gain unauthorized access to state v.

3.3 A naive attempt to build trusted FSM

The sufficient and necessary conditions in Section 3.1 for a sequential system to be trustworthy actually gives the following constructive method to build trusted FSM:

i. perform state reduction to reduce the number of states
ii. add new states $\{s1, s2, \ldots, sk\}$ to the FSM such that the total number of states becomes a power of 2
iii. add state transitions $\{s1 \rightarrow s2, s2 \rightarrow s3, \ldots, sk \rightarrow s1\}$ on any input value.
iv. for the other *don't care* transitions, make s1 as their next state
v. use minimal length codes for state encoding

Apparently, the logic implementation of the FSM following the above procedure satisfies conditions a)-c) in Section 3.1: step iv ensures that the FSM is completely specified; step ii ensures that the number of states is a power of 2; and step v requires the minimal length encoding.

The only non-trivial part of this procedure is the cycle created in step iii. By doing this, we make these new states not equivalent to each other, and thus prevent the FSM synthesis tools from merging these states. This will ensure that the total number of states is a power of 2.

From the analysis in early part of this section, we know that the FSM built by the above procedure will guarantee a trusted logic implementation of the sequential system. However, such implementation will have very high design overhead in terms of area, power and clock speed. We will report this finding in the experimental results section (Section 4).

3.4 A practical approach to building a trusted FSM

To reduce the high design overhead for building trusted FSMs, we propose a novel method that combines modification of gate level circuit and the concept of FSM re-engineering introduced in [23]. Before describing our approach, we mention that to limit the access to a protected state v, we need to protect all the states in v's starting set of states. Therefore, in the following discussion, when we consider a state to be protected, we consider all the states in its starting set of states as well.

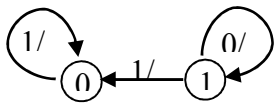


Figure 2: A simple 2-state FSM.

To illustrate the key idea of our approach, we consider a simple 2-state FSM shown in Figure 2. We assume that state 1 is the safe state and it cannot be reached from state 0. We can add a transition to enforce that the system remains in state 0 on input 0. However, for large design, adding many such

transitions will incur high overhead. Instead, we consider how to make state 1 safe at the circuit level. Without loss of generality, we assume that one T flip-flop is used to implement this system. We will use the flip flop content as a feedback signal to control the flip flop input signal (shown as the line with arrowhead in Figure 3). With the help of this new T flip flop, we see that when the system is at state 1, the feedback signal will not impact the functionality of the flip flop input signal. However, when the system is at the normal state 0, the controlled input signal will disable the T flip flop, preventing the system to go to the safe state 1.

Based on this observation, we propose to make the protected states safe by grouping them and allocating them codes with the same prefix (that is, the leading bits of the codes). For example, when the minimal code length is 5 and there are 4 states to be protected, we can reserve the 4 code words 111XX for these 4 states. Then we use flip flops with controlled signals to implement these prefix (like Figure 3 shows). The normal states (i.e., states that we do not want to control their access) will have different prefix and thus any attempt of going to a protected state from the normal states will be disabled.

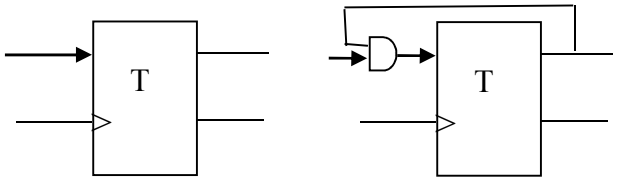


Figure 3: A normal T flip flop (on the left) and a T flip flop with controlled input (on the right).

However, when the number of states to be protected is not a power of 2, there will be unused codes with the prefix reserved for safe states. If the synthesis tools assign any of these unused code to other states, these states may gain unauthorized access to the protected states and make them unsafe. To prevent this, we apply the state duplication method proposed in [23] to introduce new states that are functionally equivalent to the safe states (see Figure 4) and mark them also as states to be protected. We repeat this until the total number of safe state becomes a power of 2.

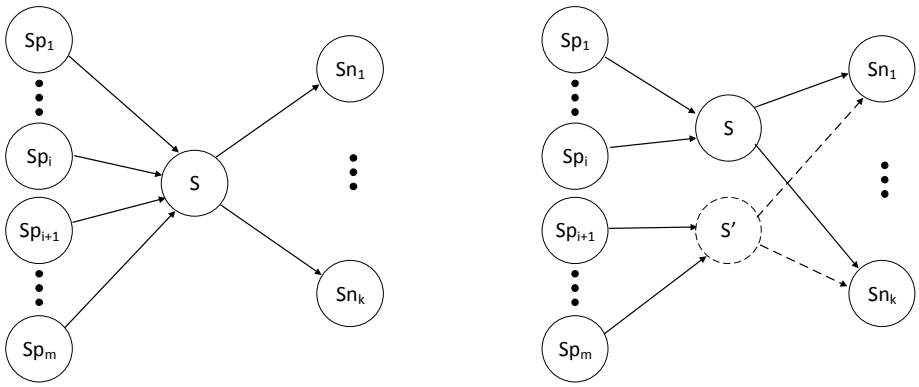


Figure 4. Reconstructing an FSM by duplicating state S [23].

Figure 4 depicts the concept of state duplication. For state S which has m previous states $Sp1, Sp2, \ldots, Spm$ and k next states $Sn1, Sn2, \ldots, Snk$ (as shown on the left), we can create a new state S' and then connecting S' to all the next states of S, but partition the previous states of S such that some of them go to the new state S' and others still go to state S. Clearly, these two FSMs are functionally equivalent. However, by doing this, S has a duplicate. If S is a state we want to protect, we need to protect its duplicate S' too. As a result, the number of states to be protected will increase. Consider an FSM with n states, $2^r - 1 < n \leq 2^r$, we apply the state duplication method to extend the total number of states that need to be protected to the nearest power of 2, 2^k. The new FSM will have no more than 2^{r+1} states. So this technique will introduce no more than one additional FF to the design. Accordingly to Yuang et al. [23], the process of state duplication gives us an opportunity to minimize power and/or area. We will report the design quality information (in terms of area, power, and delay) in the next section.

4. Experimental Results

4.1 Experimental setup and security vulnerability of current FSM synthesis flow

To validate the findings on the security risks of current sequential system design flow and the effectiveness of our proposed approach, a selection of Microelectronics Center of North Carolina (MCNC) sequential benchmark circuits, shown in Table I, in their kiss2 finite state machine format, were used. The first column gives the index for each benchmark circuit. The next 4 columns show, for each circuit, the name, the number of states, the number of input bits, and the number of transitions specified in the blif file [25]. Note that in the blif format, one transition may include multiple input values that move the current state to the next state. For example, if there is a transition from state u to state v on any of the following input values: 1000, 1001, 1010, and 1011, this will be represented in blif format by only one transition with input 10xx.

The last three columns in the table provide information for a more accurate description of each circuit. We first split each transition into edges, where each edge corresponds to only one input value. For example, the above transition on input 10xx will be split into 4 edges. The column "Number of Edges" gives the total number of edges in each circuit. From this, we can easily calculate the "Number of *don't care* edges" shown in the next column. For example, in the circuit 1 (bbara), there are 4 input bits, which means a total of 2^4=16 different input values. For each of the 10 states, there will be 16 edges, giving a total of 160 edges. The benchmark has 155 edges. So the number of *don't care* edges is 160-155 = 5. The last column gives the number of *don't care* states which can be computed as follows on the same example. We need 4 bits to encode 10 states, but 4 bits will implement 16 states, so the number of *don't care* states is 16 -10 = 6.

TABLE I. MCNC BENCHMARK CIRCUIT INFORMATION

MCNC Circuit Index	Circuit Name	Number of States	Number of Input Bits	Number of Transitions	Number of Edges	Number of Don't Cares Edges	Number of *Don't Care* States
1	bbara	10	4	56	155	5	6
2	bbsse	16	7	53	1800	248	0
3	bbtas	6	2	20	20	4	2
4	beecount	7	3	27	50	6	1
5	dk14	7	3	47	47	9	1
6	dk15	4	3	27	27	5	0
7	dk16	27	2	105	105	3	5
8	dk27	7	1	12	12	2	1
9	dk512	15	1	27	27	3	1
10	ex3	10	2	33	33	7	6
11	ex4	14	6	20	416	480	2
12	ex5	9	2	30	30	6	7
13	ex6	8	5	32	216	40	0
14	ex7	10	2	35	35	5	6

MCNC Circuit Index	Circuit Name	Number of States	Number of Input Bits	Number of Transitions	Number of Edges	Number of Don't Cares Edges	Number of *Don't Care* States
15	keyb	19	7	167	2408	24	13
16	planet	48	7	114	6016	128	16
17	S1488	48	8	250	12160	128	16
18	S1494	48	8	249	12160	128	16
19	s208	18	11	150	36352	512	14
20	sand	32	11	183	63552	1984	0
21	sse	16	7	55	1824	224	0
22	styr	30	9	164	15296	64	2
23	train11	11	2	24	24	20	5
24	train4	4	2	12	12	4	0

In most of these FSM benchmarks, each state is reachable from every other state in the FSM. There is no need to protect such states. To produce FSMs with states to be protected, we modify these benchmarks slightly by removing a small amount of transitions from the blif file so that not all of the states are reachable by all other states. In order to edit the FSMs, a program was written to read in an FSM, then remove transitions, one at a time, and record how many states have become unreachable from other states. This process is repeatable and strictly controlled to prevent an excessive number of transitions from being removed so the modified circuit can still reflect the original benchmark. The number of transitions being removed from each circuit is shown in the second column of Table II. In most cases, we only remove a couple of transitions. Similar to Table I, we expand these removed transitions and reported the number of edges being removed in the next column.

The first objective of this work is to demonstrate the vulnerability of traditional FSM synthesis and design flow. We treat states that are not reachable by some states in the FSM as states to be protected and consider all other states as normal states. We then use ABC [24] (a public logic synthesis and formal verification tool) to synthesize each of the FSMs to obtain their circuit implementation. Next we analyze these circuit implementations to generate the completely specified FSM. If the protected

states now become reachable from any normal states, these protected states will be considered unsafe. The number of such unsafe states are shown in the last column of Table II. Note that for circuits 4, 7, 15, and 19, there are no unsafe states, which means that the circuit implementation of these FSMs are trusted. However, the circuit implementations of the rest of the 20 FSMs are all untrusted.

TABLE II. FSM MODIFICATION INFORMATION

MCNC Circuit Index	Number of transitions removed	Number of edges removed	Unsafe States
1	4	5	2
2	3	56	7
3	4	4	1
4	3	3	0
5	9	9	2
6	5	5	2
7	3	3	0
8	2	2	4
9	3	3	5
10	3	3	7
11	1	32	11
12	2	2	5
13	2	32	2
14	1	1	5
15	3	24	0
16	1	128	19
17	1	128	23
18	1	128	13
19	3	512	0
20	1	1024	31
21	1	32	5

MCNC Circuit Index	Number of transitions removed	Number of edges removed	Unsafe States
22	2	48	9
23	1	1	3
24	2	2	2

Our next goal is to show that, given the vulnerability of the FSM synthesis, how attackers can gain unauthorized access to those unsafe states. For this purpose, we consider the aforementioned "random walk attack", where the attacker randomly starts with a normal state that could not reach the unsafe state in the original FSM. Then random inputs are generated so that the attacker could move around in the completely specified circuit implementation of the FSM. When the attacker reaches the unsafe stated, we mark the state as breached. For this experiment, we attempt to breach an unsafe state 10,000 times from a random starting state, and allow the attacker to generate up to 100 random inputs. For each circuit, we choose 5 unsafe states to attack. If a circuit has less than 5 unsafe states, we test all of them. Table III shows the results of our testing. For all of the rows with "n/a", those circuits do not have any unsafe states. The last column shows a very high breaching rate, 63.28% on average and close to 100% for almost half of the circuits. The three columns in the middle indicate that among the 10,000 attempts, in the worst case the attacker can succeed with only one or two input values. And in all but 4 benchmarks, the average input length to gain unauthorized access is less than 20.

TABLE III. BREACHING THE UNSAFE STATES

MCNC Circuit Index	Average number of breaches (out of 10000)	Average number of inputs	Average maximal number of inputs	Average minimal number of inputs	Average breach rate
1	913	9.68	58.50	1.50	9.13%
2	1252.6	10.29	22.80	1.60	12.53%
3	7457	2.51	17.00	1.00	74.57%
4	n/a	n/a	n/a	n/a	n/a
5	9779.5	21.15	99.00	1.00	97.80%

MCNC Circuit Index	Average number of breaches (out of 10000)	Average number of inputs	Average maximal number of inputs	Average minimal number of inputs	Average breach rate
6	10000	3.18	24.00	1.00	100.00%
7	n/a	n/a	n/a	n/a	n/a
8	10000	4.63	31.25	2.00	100.00%
9	10000	4.18	33.60	1.60	100.00%
10	8701	17.21	70.80	1.80	87.01%
11	9953.2	18.26	98.40	4.60	99.53%
12	9989.8	8.64	60.20	1.20	99.90%
13	9998	12.11	94.00	1.00	99.98%
14	9927	13.10	71.80	1.80	99.27%
15	n/a	n/a	n/a	n/a	n/a
16	9633.4	22.23	83.60	4.60	96.33%
17	5.2	6.10	23.20	3.00	0.05%
18	0	0.00	0.00	0.00	0.00%
19	n/a	n/a	n/a	n/a	n/a
20	7165.4	42.24	100.00	3.40	71.65%
21	379.2	1.49	4.80	1.20	3.79%
22	1224.8	51.72	99.20	4.60	12.25%
23	2707.67	2.29	11.67	1.33	27.08%
24	7479.5	2.02	13.50	1.00	74.80%
Average	6328.31	12.65	50.87	1.96	63.28%

4.2 Experimental results on attacks from malicious designers

A malicious designer of a sequential system has access to the original system specification and can add transitions to the blif file before FSM synthesis. (Note that we do not consider the case that the attacker removes or changes transitions from the blif file. In that case, the required

functionality of the FSM will not be implemented and such an attack can be detected relatively easily by verification tools.) For an attacker to gain unauthorized access to a protected state while hiding this malicious behavior, the attacker only need to add one transition, for example by specifying a previously *don't care* transition from a normal state to the protected state to make the state unsafe. As we have discussed earlier, a naïve way to prevent such attack is to make the FSM completely specified by specifying all the *don't care* states and the *don't care* transitions.

Tables IV-VII report the impact on design quality by this simple attack and its naïve countermeasure. We use area, power, maximal negative slack (the difference between the circuit's timing requirement and the longest delay from input to output, which measures how good the design meets its timing requirement), and the sum of negative slack as the metrics for design quality. In all of the tables, the original FSM is synthesized once to give us the baseline for comparison. We assume that the malicious attacker will add only one transition to breach the system. We allow the malicious attacker to add different transitions and design the system 10 times. The overhead of best malicious design and the average overhead over all the malicious designs compared with the baseline are reported in the next two columns. The last column is the design overhead when we apply the simple countermeasure to ensure trust in the design.

TABLE IV. AREA OVERHEAD AFTER ATTACKING UNSAFE STATES

MCNC Circuit Index	Baseline area	Overhead of the best malicious design	Average overhead of all malicious designs	Overhead of the naïve countermeasure
1	63568	0.0%	12.9%	157.7%
2	170288	-19.1%	-3.7%	155.6%
3	40368	-10.3%	0.6%	-6.9%
4	63568	1.5%	7.3%	19.0%
5	163328	-28.4%	-5.5%	-20.7%
6	79344	-6.4%	-0.4%	85.4%
7	263552	-2.3%	14.4%	33.8%
8	28304	0.0%	15.9%	52.5%
9	76096	-13.4%	3.7%	26.8%

MCNC Circuit Index	Baseline area	Overhead of the best malicious design	Average overhead of all malicious designs	Overhead of the naïve countermeasure
10	79808	-11.0%	-3.1%	79.1%
11	91872	-7.1%	13.5%	116.7%
12	69600	-5.3%	13.6%	76.7%
13	167040	-9.2%	2.9%	130.6%
14	86304	-13.4%	-0.3%	67.2%
15	284432	-17.3%	-6.5%	34.3%
16	841696	-1.5%	14.2%	68.7%
17	880208	-8.1%	9.8%	50.9%
18	889488	-2.2%	8.2%	46.3%
19	115536	-6.4%	3.7%	214.5%
20	779056	-10.9%	6.3%	84.8%
21	162400	-5.4%	7.3%	221.1%
22	934960	-31.7%	-18.9%	11.5%
23	36656	-6.3%	25.0%	215.2%
24	19952	11.6%	31.8%	209.3%
Averages:		-10.0%	5.7%	89.6%

First, as we can see from these tables, for most of the circuits, the best malicious designs have negative overhead in area, power, and slack, which means that the untrusted circuit implementations indeed have better design quality. This is not surprising because this is the best design quality when the malicious designer tries to add a single transition for each possible way to break the system. The impact on design quality by adding one transition may not be dramatic, but if the attacker designs the system multiple times, each time with a slightly different FSM, the synthesis tools may find a design with better quality. For example, in Table IV, there are only 2 circuits with area increase in the attacker's best design.

However, when we look at the data of the attacker's average design, we see an overhead of about 6% along each of the quality metrics. This result is very important in several ways. First, it shows that on average, adding even

one more transition will incur design overhead; however, the design overhead is so small that such attack cannot be detected by simply evaluating the design quality. Consider the power consumption data in Table V, only 8 out of the 24 benchmarks have more than 10% power overhead. The power overhead on other circuits might not be noticeable, and indeed there are power savings on 6 circuits.

TABLE V. POWER OVERHEAD AFTER ATTACKING UNSAFE STATES

MCNC Circuit Index	Baseline power usage	Overhead of the best malicious design	Average overhead of all malicious designs	Overhead of the naïve countermeasure
1	387	-2.3%	25.0%	202.5%
2	1250.2	-21.3%	-5.7%	147.3%
3	236.7	-7.6%	8.7%	-2.5%
4	441.1	-3.9%	6.3%	21.8%
5	1211.8	-25.3%	-3.6%	-17.0%
6	578.5	-3.7%	0.2%	86.8%
7	1999.8	0.9%	16.2%	34.2%
8	163.1	0.0%	27.5%	77.1%
9	557.6	-17.2%	2.6%	26.0%
10	592.6	-16.3%	-4.8%	83.7%
11	657.3	-6.0%	16.2%	124.5%
12	501.4	-9.5%	14.9%	74.3%
13	1293.7	-11.3%	2.1%	121.2%
14	677	-20.1%	-3.9%	55.3%
15	2066.2	-20.5%	-7.6%	36.8%
16	6520.6	-1.6%	14.1%	67.2%
17	6941.4	-11.1%	7.4%	43.4%
18	6969.3	-4.6%	6.5%	41.9%
19	790	-6.7%	8.3%	200.4%

MCNC Circuit Index	Baseline power usage	Overhead of the best malicious design	Average overhead of all malicious designs	Overhead of the naïve countermeasure
20	5939.7	-12.2%	6.4%	71.8%
21	1177.7	-9.2%	7.4%	224.9%
22	7252.3	-32.7%	-20.9%	9.5%
23	243.7	-9.7%	27.9%	249.8%
24	134.6	-0.2%	26.0%	210.8%
Averages:		-11.4%	6.8%	92.8%

Finally, the last column shows the design overhead by the simple protection mechanism. We have already analyzed in the previous section that such approach will make the FSM completely specified and thus over-constrain the design, resulting in designs with potentially very poor quality. This is confirmed in these tables. For instance, Table VI shows that the average maximal slack has increased by 44%, which means that the price we pay to ensure trustworthiness in FSM synthesis, by this approach, is probably too high. Such delay overhead may not be acceptable for many mission-critical real time embedded systems.

In summary, in 15 of the 24 circuits, the average change to the malicious circuits' statistics (area, slack, and power usage) is within ±10%. The average design overhead (see the last row of each table) is less than 8%. Furthermore, such overhead is on the sequential component of the circuit only. If we consider the entire circuit with the combinational circuitry, the overhead will become even smaller. Therefore, it will not be effective to detect malicious design by evaluating the design quality metrics. On the other hand, it is possible to apply the naïve approach by simply creating a sink state out of an unused state and redirecting all the *don't care* transitions to this sink state. This ensures the trust in the design, but the last column shows that the high performance overhead will most likely make this approach impractical.

TABLE VI. MAX NEGATIVE SLACK OVERHEAD AFTER ATTACKING UNSAFE STATES

MCNC Circuit Index	Baseline max slack	Overhead of the best malicious design	Average overhead of all malicious designs	Overhead of the naïve countermeasure
1	6.88	-9.4%	8.9%	90.4%
2	12.63	-15.4%	5.4%	77.7%
3	4.64	4.1%	14.1%	6.7%
4	7.67	-12.4%	-0.1%	22.3%
5	13.5	-23.9%	-4.6%	-7.3%
6	8.92	0.0%	6.8%	48.9%
7	19.24	-3.9%	6.6%	12.1%
8	3.75	0.0%	18.8%	45.3%
9	8.7	-17.8%	4.6%	3.6%
10	8.37	-13.1%	0.8%	42.9%
11	9.2	-9.3%	12.0%	59.6%
12	8.27	-5.2%	9.8%	22.0%
13	14.63	-8.3%	-0.8%	62.5%
14	8.56	-11.1%	5.5%	38.7%
15	17.22	-21.1%	-9.6%	5.7%
16	41.57	-2.5%	11.7%	35.7%
17	40.25	-9.1%	13.2%	29.3%
18	43.04	-9.5%	5.1%	16.9%
19	10.44	-1.0%	14.4%	69.5%
20	32.04	-12.0%	4.2%	20.8%
21	12.66	-6.8%	9.2%	129.7%
22	39.02	-24.6%	-14.6%	-2.0%
23	4.43	-10.4%	22.8%	168.4%
24	3.61	22.4%	33.7%	73.4%
Averages:		-8.8%	8.1%	44.2%

TABLE VII. SUM OF MAX NEGATIVE SLACK OVERHEAD AFTER ATTACKING UNSAFE STATES

MCNC Circuit Index	Baseline sum of max negative slack	Overhead of the best malicious design	Average overhead of all malicious designs	Overhead of the naïve countermeasure
1	32.05	-3.8%	11.7%	123.4%
2	113.19	-15.8%	-1.0%	99.3%
3	20.53	-8.5%	1.7%	-9.2%
4	40.77	-3.1%	5.4%	26.3%
5	99.34	-26.0%	-8.4%	-13.3%
6	53.19	-5.5%	0.8%	82.2%
7	124.25	5.9%	14.2%	30.4%
8	14.1	0.0%	29.2%	85.2%
9	49.34	-20.5%	1.5%	16.0%
10	44.25	-22.7%	-6.1%	63.8%
11	95.73	-11.0%	10.2%	60.5%
12	39.38	-6.1%	11.0%	62.2%
13	137.15	-9.7%	0.5%	69.6%
14	42.84	-12.0%	8.0%	59.7%
15	98.12	-17.3%	-7.3%	28.4%
16	966.4	-5.1%	9.1%	39.2%
17	919.81	-8.8%	13.7%	20.2%
18	970.58	-6.4%	6.2%	24.2%
19	56.7	1.9%	18.0%	94.1%
20	411.65	-11.4%	2.3%	32.2%
21	107.24	-4.3%	9.9%	170.8%
22	514.78	-26.5%	-15.0%	4.5%
23	15.53	-14.0%	31.6%	276.8%
24	10.42	6.8%	18.6%	173.0%

MCNC Circuit Index	Baseline sum of max negative slack	Overhead of the best malicious design	Average overhead of all malicious designs	Overhead of the naïve countermeasure
Averages:		-9.6%	6.7%	66.4%

4.3 Experimental results on the proposed practical approach

As explained in the previous section, if we do not care about the next state as long as it is not a safe state, we can use flip flops with controlled input to establish trust in the logic implementation of an FSM. To reduce the overhead caused by these controlled input signals, the FSM must be edited. If the number of safe states is less than a power of two, some of the safe states need to be duplicated using a modified version of the process in [23] to duplicate the states that will cause the least, or reduce, the overhead. The next step requires that we partially encode our FSM using a slightly modified version of the encoding algorithm. For example, if we have an FSM with 30 states and 8 states to be protected, all safe states will be given the same partial encoding in the format 11XXX. The rest of the states will have the encodings of 10XXX, 01XXX, or 00XXX. The state encoding algorithm or tool will then fill in the Xs with 0s or 1s in the most efficient manner.

Once this process is complete, the original FSM's states are encoded using the same process but all of the states start with the partial encoding comprised of all Xs. The two FSMs are then compared and the overhead is calculated for the FSM that has been modified for protection of the safe states in comparison to the original FSM. These results are reported in Table VIII. The 'encoding bit size' column shows that in most circuits, we will not increase the code length, which means no need of additional flip flops. The 'protected states' column is the percentage of the protected states, which include both the state we want to control the access and their starting set of states. The rest of the five columns report the design overhead in terms of circuit area, gate count, maximum negative slack, sum of negative slack, and power consumption.

The most important result is that in all the design quality metrics, our approach has very limited overhead, from 2.82% in the maximum negative slack to 7.49% in the gate count. More specifically, the naïve countermeasure's average overhead on circuit area, most negative slack, sum of negative slack, and power over all the benchmarks are 89.6%, 44.2%, 66.4%, and 92.8%, respectively (as reported in the last columns of Tables IV-VII). Our new approach can reduce these overhead to 7.01%, 2.82%, 5.70%, and 6.33%, respectively. Such overhead is about the same or even smaller than the average overhead that a malicious designer will have to suffer (5.7%, 6.8%, 8.1%, and 6.7% as indicated also in Tables IV-VII).

TABLE VIII. OVERHEAD FOR MANUAL ENCODING AND STATE DUPLICATION WITH CONTROLLED INPUT TO PROTECT SAFE STATES

MCNC Circuit Index	Encoding bit size	Protected States	Area	Gate Count	Most Negative Slack	Sum of Negative Slack	Power
1	0.00%	33.33%	9.49%	6.67%	9.35%	15.20%	16.15%
2	25.00%	33.33%	7.71%	6.58%	-2.01%	0.08%	6.52%
3	0.00%	0.00%	-1.20%	0.00%	2.30%	-5.66%	-2.82%
4	0.00%	33.33%	-5.17%	-12.20%	3.71%	-1.39%	2.54%
5	0.00%	33.33%	8.30%	7.14%	2.35%	0.37%	5.58%
6	50.00%	33.33%	52.13%	57.50%	32.18%	42.11%	38.43%
7	0.00%	0.00%	-11.17%	-10.76%	-9.25%	-10.68%	-11.27%
8	0.00%	33.33%	0.00%	0.00%	0.00%	0.00%	0.00%
9	0.00%	33.33%	-2.66%	-2.63%	-13.72%	-8.57%	-5.08%
10	0.00%	0.00%	53.21%	57.14%	12.69%	60.93%	49.04%
11	0.00%	33.33%	0.00%	0.00%	0.00%	0.00%	0.00%
12	0.00%	14.29%	-18.80%	-19.23%	-13.02%	-23.46%	-25.11%
13	0.00%	33.33%	7.84%	5.68%	7.42%	9.55%	9.51%
14	0.00%	0.00%	21.67%	22.86%	21.68%	26.20%	28.76%
15	0.00%	33.33%	13.95%	15.13%	14.20%	19.40%	14.01%
16	0.00%	39.13%	-6.01%	-6.73%	2.54%	2.99%	-7.23%
17	0.00%	60.00%	10.11%	8.21%	16.13%	14.13%	13.90%
18	0.00%	60.00%	3.77%	3.12%	4.15%	1.97%	4.00%
19	0.00%	0.00%	3.54%	8.70%	-1.77%	-3.12%	-2.28%
20	20.00%	6.67%	5.21%	3.91%	-3.66%	6.21%	4.43%
21	25.00%	33.33%	7.92%	6.17%	0.56%	9.35%	5.19%
22	20.00%	45.45%	-21.23%	-22.12%	-24.38%	-22.27%	-22.53%

MCNC Circuit Index	Encoding bit size	Protected States	Area	Gate Count	Most Negative Slack	Sum of Negative Slack	Power
23	0.00%	33.33%	10.95%	7.14%	7.29%	5.97%	14.15%
24	50.00%	33.33%	18.60%	37.50%	-1.11%	-2.50%	16.05%
Average	7.92%	27.45%	7.01%	7.49%	2.82%	5.70%	6.33%

5. Conclusion

Sequential systems are very important components in modern system design. Designing a trusted sequential circuit is crucial to ensure the trust of the overall system. We considered the finite state machine model of sequential circuits and defined the notions of trusted sequential system and trusted logic implementation. Then we studied several related trust issues. First, we showed that the current sequential design flow generates systems that can be easily attacked. Then we show a couple of simple and effective methods to attack these designs. Finally, we provided two constructive methods to build trusted logic implementations. The first one is a straightforward method, based on the sufficient and necessary condition for a trusted FSM, but it also introduces a high design overhead. The second approach is based on a simple circuit level modification of the flip flops and can significantly cut the overhead while also guaranteeing the trust of the FSM. We have conducted comprehensive experiments on MCNC benchmarks to validate both our findings about the vulnerability in the current FSM synthesis flow and our proposed approaches to build trust.

REFERENCES

[1] Report of the Defense Science Board Task Force on High Performance Microchip Supply, February 2005.

[2] B.S. Cohen. "On Integrated Circuits Supply Chain Issues in a Global Commercial Market –Defense Security and Access Concerns", March 2007.

[3] C.E. Irvine and K. Levitt. "Trusted Hardware: Can It Be Trustworthy?", ACM/IEEE Design Automation Conference, pp. 1-4, June 2007.

[4] S. Trimberger. "Trusted Design in FPGAs", ACM/IEEE Design Automation Conference, pp. 5-8, June 2007.

[5] G.E. Suh and S. Devadas. "Physical Unclonable Functions for Device Authentication and Secret Key Generation", ACM/IEEE Design Automation Conference, pp. 9-12, June 2007.

[6] J.A. Roy F. Koushanfar, and I. L. Markov, "EPIC: ending piracy of integrated circuits", Proceedings of the conference on Design, automation and test in Europe, pp 1069-1074, March 10-14, 2008.

[7] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Security Analysis of logic obfuscation", ACM/IEEE Design Automation Conference, pp 83-89, June 2012.

[8] J. Gu, G. Qu, and Q. Zhou. "Information Hiding for Trusted System Design", *46th ACM/IEEE Design Automation Conference (DAC'09),* pp. 698-701, July 2009.

[9] M. Banga and M. S. Hsiao, "A Region Based Approach for the Identification of Hardware Trojans," 1st IEEE International Workshop on Hardware-Oriented Security and Trust, pp.40-47, June 2008.

[10] R. Rad, M. Tehranipoor and J. Plusquellic. "Sensitivity Analysis to Hardware Trojans using Power Supply Transient Signals", 1st IEEE International Workshop on Hardware-Oriented Security and Trust, pp. 3-7 June 2008.

[11] X. Wang, M. Tehranipoor and J. Plusquellic. "Detecting Malicious Inclusions in Secure Hardware, Challenges and Solutions", 1st IEEE International Workshop on Hardware-Oriented Security and Trust, pp. 15-19, 2008

[12] Z. Gong and M. X. Makkes. "Hardware trojan side-channels based on physical unclonable functions", WISTP, pp. 294-303, 2011.

[13] S. Wei, K. Li, F. Koushanfar, and M. Potkonjak, "Hardware Trojan Horse Benchmark via Optimal Creation and Placement of Malicious Circuitry", ACM/IEEE Design Automation Conference, pp. 90-95, June 2012.

[14] A. L. Oliveira, "Techniques for the creation of digital watermarks in

sequential circuit designs," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Syst.*, vol. 20, no. 9, Sept. 2001, pp. 1101-1117.

[15] M. Lewandowski, R. Meana, M. Morrison, and S. Katkoori, "A novel method for watermarking sequential circuits," in *Proc. IEEE Int. Symp. on Hardware-oriented Security and Trust*, CA, USA, June 2012, pp. 21-24.

[16] L. Zhang and C. H. Chang, "State encoding watermarking for field authentication of sequential circuit intellectual property," in *Proc. IEEE Int. Symp. on Circuits and Syst.*, Seoul, Korea, May 2012, pp. 3013-3016.

[17] I. Torunoglu and E. Charbon, "Watermarking-based copyright protection of sequential functions," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 3, Feb. 2000, pp. 434-440.

[18] A. T. Abdel-Hamid, S. Tahar, and E. M. Aboulhamid, "A public-key watermarking technique for IP designs," in *Proc. Design, Automation and Test in Europe*, vol. 1, Munich, Germany, March 2005, pp. 330-335.

[19] A. Cui, C. H. Chang, S. Tahar, and A. T. Abdel-Hamid, "A robust FSM watermarking scheme for IP protection of sequential circuit design," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Syst.*, vol. 30, no. 5, May 2011, pp. 678-690.

[20] G. Hachtel, F. Somenzi, "Logic Synthesis And Verification Algorithms", *Kluwer Academic Publishers*, 1996.

[21] T. Kam et al., "Synthesis of FSMs: Functional Optimization", *Kluwer Academic Publishers*, 1997.

[22] C. Umans et al., "Complexity of Two-Level Logic Minimization", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* pp. 1230-1246, Vol. 25, N. 7, July 2006.

[23] L. Yuan, G. Qu, T. Villa, and A. Sangiovanni-Vincentelli. "FSM Re-Engineering: A Novel Approach to Sequential Circuit Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, Vol. 27, No. 6, pp. 1159-1164, June 2008.

[24] ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/

[25] Berkeley Logic Interchange Format (BLIF): http://www.ece.cmu.edu/~ee760/760docs/blif.pdf