REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
07-08-2023	Final Report	27-Sep-2012 - 26-Jun-2013

4. TITLE AND SUBTITLE
Final Report: Creating Digital Fingerprints in Finite State Machines

5a. CONTRACT NUMBER
W911NF-12-1-0416

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
611102

6. AUTHORS

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES AND ADDRESSES
University of Maryland - College Park
The University of Maryland
Office of Research Administration
College Park, MD 20742 -5141

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS (ES)
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSOR/MONITOR'S ACRONYM(S)
ARO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
62288-NC-II.1

12. DISTRIBUTION AVAILIBILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not contrued as an official Department of the Army position, policy or decision, unless so designated by other documentation.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	15. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Gang Qu
UU	UU	UU	UU		19b. TELEPHONE NUMBER 301-405-6703

Standard Form 298 (Rev 8/98)
Prescribed by ANSI Std. Z39.18

RPPR Final Report
as of 07-Aug-2023

Agency Code: 21XD

Proposal Number: 62288NCII **Agreement Number: W911NF-12-1-0416**
INVESTIGATOR(S):

Name: Gang Qu
Email: gangqu@mail.umd.edu
Phone Number: 3014056703
Principal: Y

Organization: **University of Maryland - College Park**
Address: The University of Maryland, College Park, MD 207425141
Country: USA
DUNS Number: 790934285 EIN: 526002033
Report Date: 26-Sep-2013 Date Received: 07-Aug-2023
Final Report for Period Beginning 27-Sep-2012 and Ending 26-Jun-2013
Title: Creating Digital Fingerprints in Finite State Machines
Begin Performance Period: 27-Sep-2012 **End Performance Period:** 26-Jun-2013
Report Term: 0-Other
Submitted By: Gang Qu Email: gangqu@mail.umd.edu
Phone: (301) 405-6703
Distribution Statement: 1-Approved for public release; distribution is unlimited.

STEM Degrees: **STEM Participants:**

Major Goals: see the uploaded report.

Accomplishments: see the uploaded report.

Training Opportunities: One Ph.D. student has been partially supported by this project.

Results Dissemination: see the uploaded report.

Honors and Awards: Nothing to Report

Protocol Activity Status:

Technology Transfer: Nothing to Report

PARTICIPANTS:

Participant Type: PD/PI
Participant: Gang Qu
Person Months Worked: 1.00 **Funding Support:**
Project Contribution:
National Academy Member: N

RPPR Final Report
as of 07-Aug-2023

Partners

,

I certify that the information in the report is complete and accurate:
Signature: Gang Qu
Signature Date: 8/7/23 3:49PM

Project Title: Creating Digital Fingerprints in Finite State Machines
Performance Period: 9/26/2012 – 6/26/2013

Abstract

The one fundamental assumption, which is normally taken for granted, for (digital) fingerprint is that there exist many or numerous objects so we can use fingerprint to distinguish or authenticate an individual from others. In human, there are billions of them. In images or multimedia contents, even more copies of the same image/video can be generated without noticeable degradation of quality. However, this may not be the case when we want to extend the application of digital fingerprint to more broad fields such as hardware, sensor nodes, message/data transmitted over a network, or more generally, the solution to a given computational hard problem. This is because that we cannot change the functionality of the hardware or the value of the data. In terms of finding solutions to a hard problem, the challenge is how to generate distinct solutions to the same problem efficiently. Without the existence of many copies, there is no room for fingerprint. What we propose here is a completely different and innovative approach: designing functional different systems or finding distinct solutions.

Objectives

The rationale behind this project is as follows: Consider a system design where we need to satisfy a set of design constraints, we explore the options/choices/freedom in the design process to implement functional different systems. These systems will all meet the set of design constraints and therefore will be acceptable. However, these systems will behave differently in certain situations based on the different requirement we impose when they are options/choices/freedom during the design process. In the 9-month period, we plan to focus on the finite state machine model, which is widely used in system design and software development, to develop the concept and demonstrate its effectiveness**.**

Findings

There are two major findings during this study as we explore the options/choices/freedom in the design process of finite state machines and circuits. First, we observe that the abundant don't care transitions and undefined states in the finite state machine are a major security and trust vulnerability, despite their ability to create numerous fingerprints (which is the objective of this project). We report this finding as well as the mitigation methods in a journal article [1] and a book chapter [2]. We did not investigate further on the creating digital fingerprints in finite state machine using these don't care conditions because the difference at finite state machine level will require redesign at circuit level and thus result in different masks for semiconductor fabrication, which are prohibited expensive.

The second finding is motivated by this as we look for more practical fingerprinting solutions. As a result, we propose, for the first time, two post-silicon fingerprinting approaches based on don't care conditions. The results are published in two top-tier conferences [3,4] and included in a book chapter on digital fingerprinting [5].

PDF of references [1], [3], and [4] are attached in this report. The two book chapters are available upon request.

[1] Carson Dunbar and Gang Qu. Designing trusted embedded systems from finite state machines, ACM Transactions on Embedded Computing Systems (TECS) 13 (5s), pp. 1-20. 2014.

[2] Carson Dunbar and Gang Qu. *Towards Building Trusted Systems: Vulnerabilities, Threats, and Mitigation Techniques*, in Secure System Design and Trustable Computing, Springer, pp. 301-328. 2016.

[3] Carson Dunbar and Gang Qu. Satisfiability don't care condition based circuit fingerprinting techniques, 20th IEEE Asia and South Pacific Design Automation Conference, pp. 815-820. 2015.

[4] Carson Dunbar and Gang Qu. A practical circuit fingerprinting method utilizing observability don't care conditions, 52nd ACM/IEEE Annual Design Automation Conference, pp. 1-6. 2015.

[5] Gang Qu, Carson Dunbar, Xi Chen and Aijiao Cui. *Digital fingerprint: A practical hardware security primitive*, In Digital Fingerprinting, Springer, pp. 89-114. 2016.

Designing Trusted Embedded Systems from Finite State Machines

CARSON DUNBAR and GANG QU, University of Maryland, College Park

Sequential components are crucial for a real-time embedded system as they control the system based on the system's current state and real life input. In this article, we explore the security and trust issues of sequential system design from the perspective of a finite state machine (FSM), which is the most popular model used to describe sequential systems. Specifically, we find that the traditional FSM synthesis procedure will introduce security risks and cannot guarantee trustworthiness in the implemented circuits. Indeed, we show that not only do there exist simple and effective ways to attack a sequential system, it is also possible to insert a 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 Microelectronics Center of North Carolina (MCNC) sequential circuit benchmarks.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Design, Security

Additional Key Words and Phrases: Finite state machine, EDA, flip-flop, trust, unauthorized access

ACM Reference Format:
Carson Dunbar and Gang Qu. 2014. Designing trusted embedded systems from finite state machines. ACM Trans. Embedd. Comput. Syst. 13, 5s, Article 153 (September 2014), 20 pages.
DOI: http://dx.doi.org/10.1145/2638555

1. INTRODUCTION

As electronic design automation (EDA) and semiconductors continue to evolve rapidly, a company will not typically have all the expertise and capability to do in-house design and to 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 [Defense Science Board 2005; Cohen 2007; Irvine and Levitt 2007; Trimberger 2007; Suh and Devadas 2007; Roy et al. 2008; Rajendran et al. 2012; Gu et al. 2009].

When the company gives the system's specifications 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 specifications. An adversary can simply

This work is supported by the National Natural Science Foundation of China under Grant No. 61228204, Army Research Office under grant W911NF1210416, Air Force Office of Scientific Research under grant FA95501010140, a university partnership with Laboratory for Telecommunications Sciences (H9823013D00560002), and Air Force Research Laboratory under agreement number FA8750-13-2-0115.
The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government.
Author's address: C. Dunbar and G. Qu, Electrical and Computer Engineering Department, University of Maryland, College Park; email; gangqu@glue.umd.edu.

DOI: http://dx.doi.org/10.1145/2638555

add additional circuitry, known as a hardware Trojan horse, to maliciously modify the system. For example, a Trojan 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 [Banga and Hsiao 2008; Rad et al. 2008; Wang et al. 2008; Gong and Makkes 2011; Wei et al. 2011].

In this article, 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 one 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: these 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 $x = 1$.

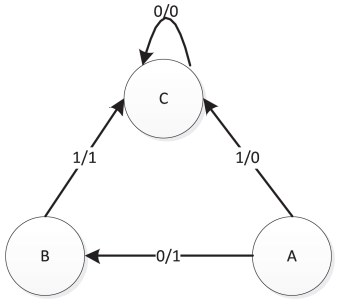
Second, when both flip flops have value 1, the system will be in a state that is not specified in the original FSM. With input $x = 0$ and$x = 1$, the system will output 1 and move to state C and state A, respectively. In other words, the full functionality of the circuit in Figure 1(b) can be described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

The FSM in Figure 1(a) is what we want to design and 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.

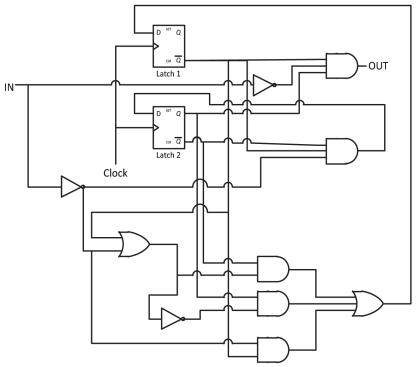
In the original FSM (Figure 1(a)), when the system leaves state A, it cannot come back. In other words, we say that there is no access to state A from state B and state C. However, in the implemented FSM in Figure 1(c), each state has at least one path to come back to state 00. For instance, from state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

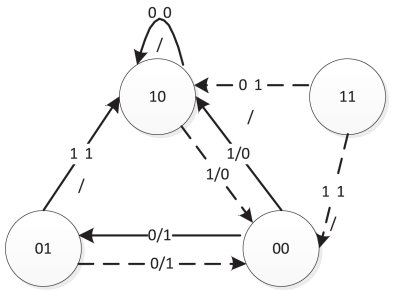
The challenge of building trust into an IC is highlighted in a Defense Science Board study on High Performance Microchip Supply, "Trust cannot be added to integrated circuits after fabrication; electrical testing and reverse engineering cannot be relied on to detect undesired alternations in military integrated circuits" [Defense Science Board 2005]. The Trusted Foundry Program and the complementary Trusted IC Supplier



(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).

Fig. 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).

Accreditation were specifically designed for domestic fabrication, where trust and accreditation are built on reputation and partnership [Cohen 2007]. The notions of trusted and trustworthiness are presented in Irvine and Levitt [2007], where the authors discuss the challenges, opportunities, call for new initiatives and programs to establish principles, tools, and standards for building trusted hardware. Trimberger [2007] 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 [2007] 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 ICs [Roy et al. 2008; Rajendran et al. 2012]. However, none of these efforts can be used to verify whether the chip contains unwanted functionalities. Gu et al. [2009] develop an 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 novel concept is hard to be implemented due to the complexity of the 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 [2008] propose a circuit partition-based approach to detect and locate the embedded Trojan. Rad et al. [2008] 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. [2008] 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 [2011] present a Trojan side-channel-based on PUF that can successfully attack block ciphers [Gong et al. 2011]. Wei et al. [2012] 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 [Oliveira 2001; Lewandowski et al. 2012; Zhang and Chang 2012; Torunoglu and Charbon 2000; Abdel-Hamid et al. 2005; Cui et al. 2011]. These techniques usually rely on the modification of the state transition graph at the behavioral synthesis level to embed a watermark related to user-specific information for identification purposes. Oliveira proposes creating watermarks based on a set of redundant states that can only be traversed when a user-specific input sequence is loaded [Oliveira 2001]. Lewandowski et al. [2012] and Zhang and Chang [2012] propose watermarking schemes based on state encoding. Torunoglu and Charbon [2000] introduce extra state transitions in the FSM to produce output that carries the watermark. Abdel-Hamid et al. [2005] improve this method by utilizing the existing transitions for watermarking and successfully reduce the high-overhead caused by extra state transitions. Cui et al. [2011] 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).

In this article, we consider the security and trust vulnerabilities in the current sequential system design and implementation flow. These vulnerabilities exist in the high-level incomplete system specification and become real threats during system implementation. We will analyze how an adversary can attack such untrusted systems in two different scenarios. In the first case, the attacker attempts to attack a system that is already implemented by exploiting potential trapdoors. The trapdoors in the IC are additional functionalities inadvertently implemented. They are conceptually different from hardware Trojans or watermarks, which are added intentionally. In the second

case, the attacker has access to the system during its design and implementation process and can maliciously alter the system specification and/or implementation. This can be considered as adding hardware Trojans by watermarking. However, because the Trojans are introduced early in the system specification and will be integrated into the design, the previously mentioned Trojan detection techniques based on IC abnormality will fail to identify such attacks. Therefore, we need to study both attacking scenarios and develop new countermeasures.

The rest of the article is organized as follows. In Section 2, we give the necessary background of finite state machines. 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;
—$\delta : S \times I \rightarrow S$ is the next-state function;
—$S0 \subseteq S$ is the set of initial states;
—O is the output alphabet;
—$\lambda : S \times I \rightarrow O$ 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 a graph is often referred to as a *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

$$R(u) = \{v \in V | v \textit{ is reachable from } u\},$$

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

$$S(u) = \{v \in V | u \textit{ is reachable from } v\},$$

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 a minimal number of states is generally referred to as a *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 [Hachtel and Somenzi 1996]. For incompletely specified machines, although the problem is NP-hard, there are standard approaches to solve the state reduction problem [Kam et al. 1997].

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 [Umans et al. 2006]. 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 COUNTERMEASURES

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 phases 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 purposes 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 it will produce an illegal entry to that protected state (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 this discussion, we conclude the following.

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 preceding analysis shows that all the three conditions are necessary. To see they are also sufficient, condition (a) indicates that there are no *don't care* transitions; conditions (b) and (c) guarantee that in the implementation of the system there are 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 systems it is impossible to completely specify a system's behavior with 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 the 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) $\Leftrightarrow$ (2) $\Leftrightarrow$ (3). Since (1) is the conjunction of (2) and (3), it suffices to show that (2) $\Leftrightarrow$ (3). We prove (2) $\Rightarrow$ (3) by contradiction as follows. (3) $\Rightarrow$ (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 a certain unreachable state without being detected. In this case, the attacker can implement such a 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 (a state 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 nonminimal length encoding is used). He can simply launch an 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 preceding procedure satisfies conditions (a)–(c) in Section 3.1. Step (4) ensures that the FSM is completely specified; step (2) ensures that the number of states is a power of 2; and step (5) requires the minimal length encoding.

The only nontrivial part of this procedure is the cycle created in step (3). 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 the early part of this section, we know that the FSM built by the this 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 the gate level circuit and the concept of FSM re-engineering introduced in Yuan et al. [2008]. Before describing our approach, we mention that to limit 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.

To illustrate the key idea of our approach, we consider the 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 a 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 from going to safe state 1.

Based on this observation, we propose making the protected states safe by grouping them and allocating codes to them with the same prefix (that is, the leading bits of the codes). For example, when the minimal code length is five and there are four states to be protected, we can reserve the four code words 111XX for these four states. Then we use flip-flops with controlled signals to implement these prefix as (as shown in

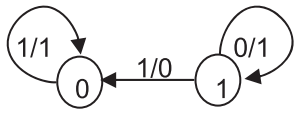


Fig. 2. A simple 2-state FSM.

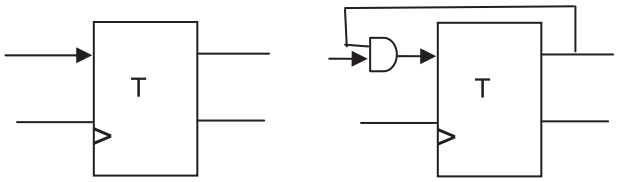


Fig. 3. A normal T flip-flop (on the left) and a T flip-flop with controlled input (on the right).

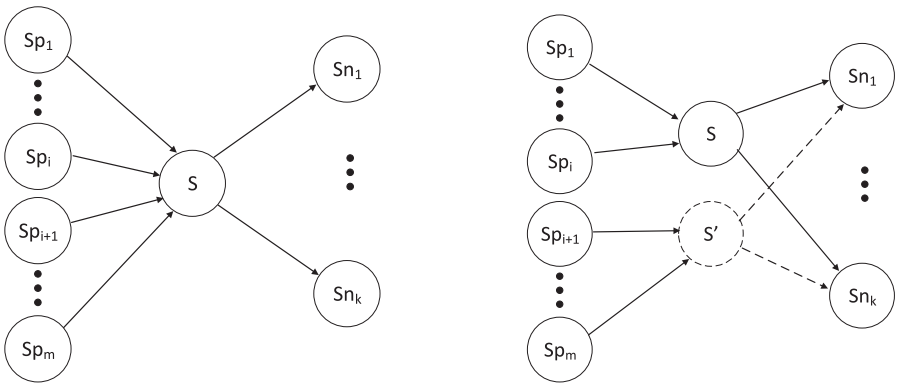


Fig. 4. Reconstructing an FSM by duplicating state S [Yuan et al. 2008].

Figure 3). The normal states (states for which we do not want to control access) will have different prefixes and thus any attempt to go to a protected state from the normal state will be disabled.

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 codes 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 Yuan et al. [2008] 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 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 connect 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 also need to protect its duplicate S. 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

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
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

than one additional FF to the design. As in Yuang et al. [2008] 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 four 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 [BLIF]. 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 preceding 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 circuit 1 (bbara), there are four input

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
22	2	48	9
23	1	1	3
24	2	2	2

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 that can be computed as follows on the same example. We need four bits to encode 10 states, but four bits will implement 16 states, so the number of *don't care* states is $16 - 10 = 6$.

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 number 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 report 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 [ABC] (a public logic synthesis and formal verification tool) to synthesize each of the FSMs to obtain its circuit implementation. Next we analyze these circuit

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%
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%

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 is 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 implementations of these FSMs are trusted. However, the circuit implementations of the rest of the 20 FSMs are all untrusted.

Our next goal is to show 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 state, we mark it 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 five unsafe states to attack. If a circuit has less than five 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

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%
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%

values. And in all but four benchmarks, the average input length to gain unauthorized access is less than 20.

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 where 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 needs 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 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

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%
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%

attacker to add different transitions and design the system 10 times. The overhead of the 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.

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 two 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 an attack cannot be detected by simply evaluating the design quality. Consider the power consumption data in Table V, only eight out of the 24 benchmarks have more than 10%

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%

power overhead. The power overhead on other circuits might not be noticeable, and indeed there are power savings on six circuits.

Finally, the last column shows the design overhead by the simple protection mechanism. We have already analyzed in the previous section that such an 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 trust in the design, but the last column shows that the high performance overhead will most likely make this approach impractical.

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%
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 Yuan et al. [2008] to duplicate the states that will cause the least, or reduced, 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 eight states to be protected, all safe states will be given the same partial encoding in 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, although 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 to and the starting set of the states.

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%
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%

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 overheads 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 would have to suffer (5.7%, 6.8%, 8.1%, and 6.7% as also indicated in Tables IV–VII).

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 necessary and sufficient 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 building trust.

REFERENCES

ABC: A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~alanmi/abc/.

A. T. Abdel-Hamid, S. Tahar, and E. M. Aboulhamid. 2005. A public-key watermarking technique for IP designs. In *Proceedings of Design, Automation and Test in Europe*. 330–335.

M. Banga and M. S. Hsiao. 2008. A region based approach for the identification of hardware Trojans. In *Proceedings of First IEEE International Workshop on Hardware-Oriented Security and Trust*. 40–47.

BLIF. Berkeley gic Interchange Format. http://www.ece.cmu.edu/~ee760/760docs/blif.pdf.

B. S. Cohen. 2007. On integrated circuits supply chain issues in a global commercial market—Defense security and access concerns. Information Technology and Systems Division, Institute for Defense Analyses (Statement before US House of Representatives Armed Services Subcommittee 3/14/07).

A. Cui, C. H. Chang, S. Tahar, and A. T. Abdel-Hamid. 2011. A robust FSM watermarking scheme for IP protection of sequential circuit design. *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.* 30, 5, 678–690.

Defense Science Board 2005. Report of the defense science board task force on high performance microchip supply.

Z. Gong and M. X. Makkes. 2011. Hardware Trojan side-channels based on physical unclonable functions. In *Proceedings of WISTP*. 294–303.

J. Gu, G. Qu, and Q. Zhou. 2009. Information hiding for trusted system design. In *Proceedings of 46th ACM/IEEE Design Automation Conference (DAC)*. 698–701.

G. Hachtel and F. Somenzi 1996. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers.

C. E. Irvine and K. Levitt. 2007. Trusted hardware: Can it be trustworthy? In *Proceedings of ACM/IEEE Design Automation Conference*. 1–4.

T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. 1997. *Synthesis of FSMs: Functional Optimization*. Kluwer Academic Publishers

M. Lewandowski, R. Meana, M. Morrison, and S. Katkoori. 2012. A novel method for watermarking sequential circuits. In *Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust*. 21–24.

A. L. Oliveira. 2001. Techniques for the creation of digital watermarks in sequential circuit designs. *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.* 20, 9, 1101–1117.

R. Rad, M. Tehranipoor, and J. Plusquellic. 2008. Sensitivity analysis to hardware Trojans using power supply transient signals. In *Proceedings of the 1st IEEE International Workshop on Hardware-Oriented Security and Trust*. 3–7.

J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri. 2012. Security analysis of logic obfuscation. In *Proceedings of the ACM/IEEE Design Automation Conference.* 83–89.

J. A. Roy, F. Koushanfar, and I. L. Markov. 2008. EPIC: Ending piracy of integrated circuits. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 1069–1074.

G. E. Suh and S. Devadas. 2007. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the ACM/IEEE Design Automation Conference.* 9–12.

I. Torunoglu and E. Charbon. 2000. Watermarking-based copyright protection of sequential functions. *IEEE J. Solid-State Circuits*, 35, 3, 434–440.

S. Trimberger. 2007. Trusted design in FPGAs. In *Proceedings of the ACM/IEEE Design Automation Conference*. 5–8.

C. Umans, T. Villa, and A. Sangiovanni-Vincentelli. 2006. Complexity of two-level logic minimization. *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.* 25, 7, 1230–1246.

X. Wang, M. Tehranipoor, and J. Plusquellic. 2008. Detecting malicious inclusions in secure hardware, challenges and solutions. In *Proceedings of the 1st IEEE International Workshop on Hardware-Oriented Security and Trust*. 15–19.

S. Wei, K. Li, F. Koushanfar, and M. Potkonjak. 2012. Hardware Trojan horse benchmark via optimal creation and placement of malicious circuitry. In *Proceedings of the ACM/IEEE Design Automation Conference*. 90–95.

L. Yuan, G. Qu, T. Villa, and A. Sangiovanni-Vincentelli. 2008. FSM re-engineering: A novel approach to sequential circuit synthesis. *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.* 27, 6, 1159–1164.

L. Zhang and C. H. Chang. 2012. State encoding watermarking for field authentication of sequential circuit intellectual property. In *Proceedings of the IEEE International Symposium on Circuits and Systems*. 3013–3016.

Received April 2013, November 2013; revised March 2014; accepted June 2014

Satisfiability Don't Care Condition Based Circuit Fingerprinting Techniques

Carson Dunbar and Gang Qu
Department of Electrical and Computer Engineering
University of Maryland, College Park, MD 20742
{cdunbar, gangqu}@umd.edu

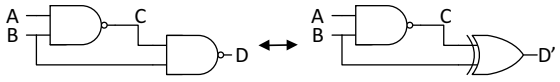
***Abstract* – Circuit fingerprints allow the authors of design intellectual properties (IPs) to trace each copy of their IPs by embedding features, known as digital fingerprints, which are unique to each device. In this paper, we propose a novel gate replacement approach to encode fingerprints based on the inherent Satisfiability Don't Care (SDC) conditions in the circuit. Moreover, existing fingerprinting schemes all require redesign of the circuit which makes it prohibitively expensive for manufacturing. We develop a practical method to implement our SDC-based circuit fingerprint. First, we introduce flexibilities during the logic synthesis phase by replacing certain library cells with versatile multiplexers (MUXs). The MUX can be configured either as the original gate or one of its replacements with identical functionality except the SDC conditions. Then at the post-silicon stage, we configure these MUXs to create distinct fingerprints. We consider standard benchmark circuits and demonstrate that even on these circuits with limited size, we can find sufficient locations to embed fingerprints. Simulation with TSMC 0.35µm technology shows non-trivial design overhead, however, such overhead will become negligible for large real-life circuits.**

I. 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].

Traditionally, design information was protected from theft by law enforcement mechanisms such as patents, copyrights, and encryption methods. Recently, digital watermarking and fingerprinting have been shown to be two additional effective IP protection techniques. Watermarking enables the designer of the IP to prove the authorship of an IP, while fingerprinting allows the tracking of each individual sold IP. Therefore, when the IP designer suspects IP piracy, he can verify this by retrieving the embedded watermark, and then use the fingerprint to locate the source of the IP piracy (i.e., which IP buyer has been involved in the IP piracy).

For this purpose, an IP fingerprinting technique needs to meet the following three fundamental requirements: (1) Correct functionality: Failure to provide the desired functionality will make the IP useless; (2) Distinct fingerprints: If two copies of the IP have identical fingerprints, then we won't be able to tell which one is the source of the IP piracy; (3) Heredity: The fingerprint must stay in any illegally reproduced IP instances in order to enable the trace of IP piracy. As we will survey in the related work section, current circuit fingerprinting methods either violate these requirements or require non-trivial (re-)design efforts.



(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)

Fig. 1. Example SDC modification (a) Gate Replacement (b) Truth table for both circuits

In this paper 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. Fig. 1 shows a trivial example of this.

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.

II. RELATED WORK

IP protection is becoming an increasingly relevant topic in the field of electronic design automation (EDA). This is because the number of ICs being manufactured by firms that designed them is shrinking. With many parties being involved in the production of a single circuit, it is easier for someone to steal information about the circuit for their own benefit. Two major techniques have been developed to counteract this, watermarking and fingerprinting, which this section will discuss.

A. *Circuit Watermarks*

The first main method of IP protection is embedding a watermark into a circuit. This concept is similar to implanting

This work is supported in part by the National Natural Science Foundation of China under Grant No. 61228204

978-1-4799-7792-5/15/$31.00 ©2015 IEEE

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.

One of the older methods of watermarking for integrated circuits (ICs), developed by Chabon in [5], 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 [3], which gives techniques for forcing specific characteristics onto the circuit from the design process.

The work in [6] 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.

Cui and Chang [1] propose a watermarking scheme at logic synthesis level based on circuit replacement. In their approach, small sub-circuits are identified and redesigned to embed the digital watermark. These redesigned sub-circuits are functionally identical to the original ones, but most likely have different layout due to redesign. Our gate replacement technique is conceptually different: it works at individual logic cell level, does not require redesign, does not change the circuit layout, and more importantly, does not maintain the complete functional equivalence (except on the conditions that are not SDC).

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.

B. Circuit Fingerprints

Circuit fingerprinting attempts to fix the issue of single use watermarks by using individual characteristics of a circuit. Like 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.

Patel et al. created a technique in [7] 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.

Jin et al., from [8], use a different technique that takes advantage of process variations, as opposed to glitches that can be fixed. They measure various path delays in their ICs with significant accuracy to create a fingerprint for a circuit. These path delays vary enough to give each IC a unique signature. This helps them prevent Hardware Trojans, which would increase the path delay and change the output of the IC in a harmful way.

Both [7] and [8] are techniques for fingerprinting individual circuits utilizing process variations. Neither technique is suitable for IP protection because if an adversary is willing to duplicate an IC, the fingerprint will be completely different from the fingerprint in the original IC. If this is the case, someone trying to prevent duplication would not be able to trace the adversary copying their IP because they would not have a record of the duplicated IC's fingerprint.

Due to its difficulty, fingerprinting of ASICs for IP protection does not have as much active or completed research as watermarking. The first major paper on this specific topic was done by Caldwell et al. of [2], 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 [9, 4, 10], with different criteria to create independent fingerprints for each buyer of the device.

Building off the idea in [2], a conceptually different fingerprinting method for the graph coloring problem is proposed in [11], 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 [2] if the design was highly specified before optimization.

The issue with both [2] and [11] 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 we propose to make our contribution and improvements to the state-of-the-art.

Finally, we mention the emerging technology of physical unclonable functions (PUFs). 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 bit stream as the cryptographic key. PUFs are built separated from ICs that utilize them and thus they do 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, PUFs

can be used as a fingerprint for device authentication, but not for IP protection.

III. 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 [2, 11].

A. 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 Fig. 1, *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

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

In the example of the above 2-input NAND gate, the SDC conditions can be obtained from Equation (1) as follows:

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

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 Fig. 1.

B. 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).

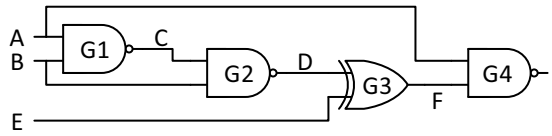


Fig. 2: Example of a dependent line. B is a dependent line for gate G2 and A is a dependent line for G4.

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.

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 Fig. 2, 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 Fig. 2, 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 Fig. 2, 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 G1, G2, …, Gn;
2.  for each gate Gi (i=1, 2, …, n)
3.  { if (Gi has a dependent line)
4.    { find the cone rooted at Gi whose inputs are the dependent
      line or the intermediate lines closest to Gi;
5.      for each combination of Gi's fan-ins that does not happen
6.      { mark Gi 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 Gi'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.

C. 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.

Fig. 1 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 [12], 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. Fig. 3 illustrates how the fuses will be used to implement the fingerprint.

IV. 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.

A. 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.

B. 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.

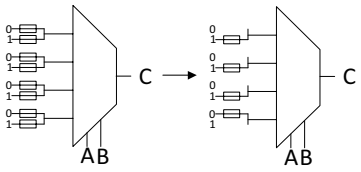


Fig. 3. Multiplexer replacement technique. Left: An unconfigured MUX; Right: A MUX configured to run as a 2-input NAND gate.

C. 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.

V. 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 [13], ISCAS-89 [14], and MCNC [15] benchmark libraries, seen under the *Unmodified Circuit Information* section of Table I. The library used for delay and area measurements is the Oklahoma State University standard cell library based on the TSMC 0.35µm technology [16]. Measurements for area and delay were collected using the program ABC [17].

A. Fingerprint Potential

The maximum size of the fingerprints we were able to find for the replacement methods **R1** and **R2** are given in Table I. 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 I 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 I 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.

B. 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 I 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 Fig. 4 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).

TABLE I. 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*	*Maximum Bits Found*	*Maximum 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%

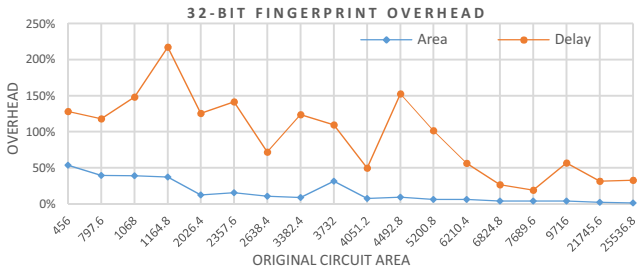


Fig. 4: Overhead results for 32-bit fingerprint using multiplexer replacement.

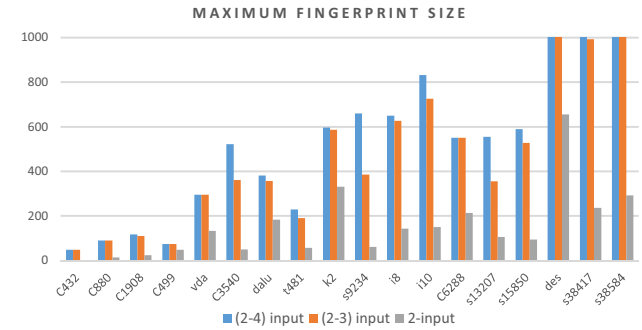


Fig. 5. Number of bits found for a 1024 bit fingerprint with gate replacement limitations

C. *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 Fig. 5.

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.

VI. 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.

VII. Acknoledgement

The authors are supported by Army Research Office under grant W911NF1210416, a university partnership with Laboratory for Telecommunications Sciences (H9823013D00560002), and Air Force Research Laboratory under agreement number FA8750-13-2-0115.

VIII. 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] A. E. Caldwell and e. al., "Effective Iterative Techniques for Fingerprinting Design IP," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, New York, NY, 1999.

[3] A. B. Kahng and e. al., "Robust IP watermarking methodologies for physical design," in *Proceedings of the ACM/IEEE Design Automation Conference*, Piscataway, NJ, 1998.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] A. B. Kahng and e. al., "Copy Detection for Intellectual Property Protection of VLSI Desgin," in *IEEE/ACM International Conference on Computer-Aided Design*, San Jose, California, 1999.

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

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

[12]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.

[13]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.

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

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

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

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

A Practical Circuit Fingerprinting Method Utilizing Observability Don't Care Conditions

Carson Dunbar and Gang Qu
Electrical and Computer Engineering Department and Institute for Systems Research
University of Maryland, College Park, Maryland, USA
{cdunbar, gangqu}@umd.edu

***Abstract*— Circuit fingerprinting is a method that adds unique features into each copy of a circuit such that they can be identified for the purpose of tracing intellectual property (IP) piracy. It is challenging to develop effective fingerprinting techniques because each copy of the IP must be made different, which increases the design and manufacturing cost. In this paper, we explore the Observability Don't Care (ODC) conditions to create multiple fingerprinting copies of a design IP (e.g. in the form of gate level layout) with minute changes. More specifically, we find locations in the given circuit layout where we can replace a gate with another gate and some wires without changing the functionality of the circuit. However, as expected, this could introduce design overhead. Our experimental results show that, although we can embed fingerprints of up to 1438 bits, there is an average of 10.9% area increase, 50.5% delay increase, and 9.4% power increase on circuits in the MCNC and ISCAS 85 benchmark suites. We further propose a fingerprinting heuristics under delay constraints to help us reduce area and power overhead.**

I. INTRODUCTION

With the system on a chip (SoC) paradigm becoming more popular, especially for small embedded system developers, intellectual property (IP) blocks such as cores and memory units are becoming essential. These IPs enable the more efficient reuse based design methodology [1]. However, it also makes IP theft a profitable business and a major threat to IP developers, vendors, and SoC industry in general and hence motivates the IP protection problem [1, 2, 7, 10].

In addition to law enforcement mechanisms, such as patent, copyright, and encryption, watermarking and fingerprinting have proven to be effective countermeasures to theft. Watermarking enables the designer of the IP to prove the authorship of an IP, while fingerprinting allows the tracking of each individual sold IP. Therefore, when the IP designer suspects any IP piracy, he can verify this by retrieving the embedded watermark, and then use the fingerprint to locate the source of the IP piracy (i.e., which IP buyer has been involved in the IP piracy).

For this purpose, an IP fingerprinting technique needs to meet the following three fundamental requirements: (1) *correct functionality*. Failure to provide the desired functionality will make the IP useless; (2) *distinct fingerprints*. If two copies of the IP have identical fingerprints, then we won't be able to tell which one is the source of the IP piracy; (3) *heredity*. The fingerprint must stay in any illegally reproduced IP instances in order to enable the trace of IP piracy. As we will survey in the related work section, current circuit fingerprinting methods either violate these requirements or require non-trivial (re)design efforts.

DAC '15, June 07 - 11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3520-1/15/06…$15.00
http://dx.doi.org/10.1145/2744769.2744780

A. Improvements to the State of the Art

In this paper, we propose a practical method to generate a large number of distinct fingerprinted IPs with little design and re-design overhead. Our method utilizes a two-step process to introduce fingerprints to individual integrated circuits (ICs). First, an IC is designed with a number of flexibilities so every IC fabricated is identical. Second, in the post-silicon stage, the flexibilities are solidified such that each IC has an individual fingerprint. Introducing flexibility in circuits reduces the redesign for fingerprints by moving fingerprint application to the last stages of the VLSI design cycle.

We first use a small example to show the basic ideas behind our fingerprinting approach. Then we review the related work and provide the background in section II. The proposed method is elaborated in section III and our experimental setup is explained in section IV. Finally, we report the experimental results and conclude the paper in sections V and VI.

Illustrative example.

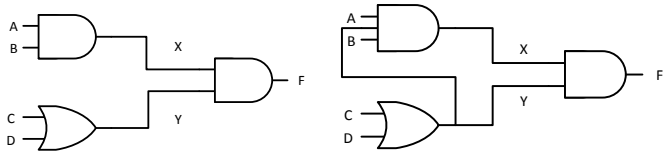


Fig 1. Two 4-input circuits that implement the same function.

The left circuit in Figure 1 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 1, 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 1, 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 [2, 3].

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 2 also implement the same function F as those circuits in Figure 1 and they all have similar layout. The contribution of this paper is the discovery of the proposed practical fingerprinting method and its implementation.

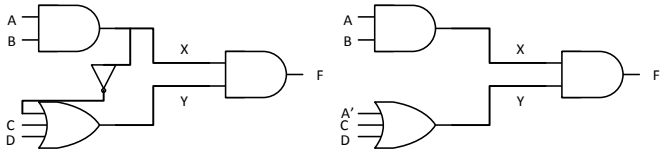


Fig 2. Two more implementation of the same function.

The first major challenge was 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 at 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, with 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.

II. RELATED WORK

IP protection is becoming a more relevant topic in circuit design as many circuits are not being manufactured by the firms that designed them, as they used to be. With many parties being involved with the production of a single circuit, it is easier for someone to steal information about the circuit for their own benefit. In this section we will discuss the current work being done on circuit fingerprinting.

A. Circuit Fingerprints

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.

Patel et al. created a technique in [2] 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.

Jin et al., from [3], use a different technique that takes advantage of process variations as opposed to glitches that can be fixed. They use the delay path variations 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.

Both [2] and [3] are techniques for fingerprinting individual circuits. Neither technique is 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.

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 [4], 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 [5], [6], [7], and [12], with different criteria to create independent fingerprints for each buyer of the device.

Building off the idea of the idea in [4], a conceptually different fingerprinting method for the graph coloring problem is proposed in [8], 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 [4] if the design was highly specified before optimization.

The issue with both [4] and [8] 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 we propose to make our contribution and improvements to the state-of-the-art, in a similar manner to the work in [9].

Finally, we mention the physical unclonable function (PUF), a new security technology. 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. PUFs are generated in additional circuitry and thus they 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.

III. OBSERVABILITY DON'T CARE (ODC) FINGERPRINTING

The goal of this work is to define a new paradigm of fingerprinting that depends on IC post-silicon flexibility and improves on previous work in the field. Most current fingerprinting techniques, such as the work in [2] and [3], 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 techniques rely on process variation 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 [4] and [8] 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.

A. Observability Don't Cares

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 3 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.

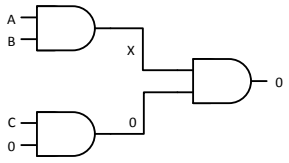


Fig 3. ODC on the AND gate

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:

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

Basically, this 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 3), input *x* becomes an ODC.

B. 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 1 (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 I) 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 1 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.

C. 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 4.

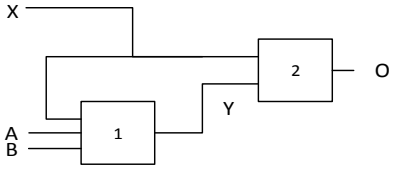


Fig 4. Generic fingerprint change

Figure 4 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 4, 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 2 (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 ODCs and single input gates in a library. An example of this exists for the library we used, in Table II, later in this section. 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 4 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 4, a fingerprint location may also consist of more than one input gate as stated in Definition 1. If this is the case, a modification similar to the one in Figure 4 may be applied to each of the input gate in the FFC that abides by Definition 1, 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 5 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 5 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 5 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.

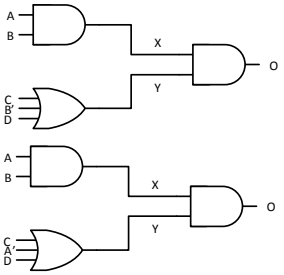


Fig 5. Fingerprint change that reroutes earlier signals

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.

D. 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.

E. 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 IP piracy has occurred. 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 1 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. Carefully designed fingerprint copy distribution schemes may help [2, 3, 8], but require a large number 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 [2, 3, 8].

IV. 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) and ISCAS '85 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 [10] 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.

A. Initial Fingerprinting

Our first goal was to create a program that could implement the details of section III 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.

Combining this information with the ODC calculations, done with equation 1 and the gates in our library, allowed us to determine what gate combinations were viable fingerprint locations.

```
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
```

Fig 6. 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 6 shows this entire process in pseudo-code.

Each of benchmark circuit netlists were run through our circuit modifier and the resulting performance metrics (area, delay, and power) were recorded. Table II 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.

B. Reduction of Overhead

Our original results, shown in Table III, 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 III C, 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.

V. 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 II 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 II 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 III. These configurations can increase the total number of fingerprints significantly, as can be seen in the column six of Table II. Column 7 of Table II 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.

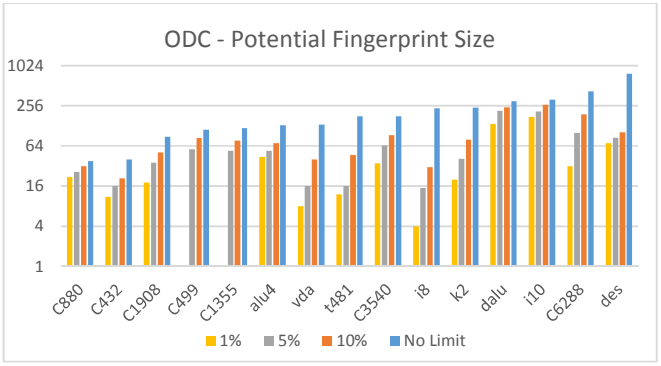


Fig 7. Fingerprint sizes for tested circuits before and after constraints

TABLE II. 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%

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 III 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 7 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 III. 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%*

VI. CONCLUSION

This work has shown that we can create a significant number of fingerprints that are hard to detect, and thus hard to remove even for the relatively small benchmark circuits. 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 are several potential directions to continue this work. One of the most important and interesting one is to investigate how to implement these fingerprints further down the manufacturing line to make them practical for real life designs. 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.

VII. ACKNOWLEDGEMENT

This work is supported in part by the National Natural Science Foundation of China under Grant No. 61228204 and by the Army Research Office under grant W911NF1210416.

VIII. 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] 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.

[3] 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.

[4] 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.

[5] A. B. Kahng and e. al., "Copy Detection for Intellectual Property Protection of VLSI Desgin," in *IEEE/ACM International Conference on Computer-Aided Design*, San Jose, California, 1999.

[6] 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.

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

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

[9] C. Dunbar and G. Qu, "Satisfiability Don't Care Condition Based Circuit Fingerprinting Techniques," in *Proceedings of 20th Asia and South Pacific Design Automation Conference (ASP-DAC), 2015* , Chiba, Japan, 2015.

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

[11] 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.

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

[13] 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.

[14] A. T. Abdel-Hamid, S. Tahar and E. M. Aboulhamid, "A survey on IP watermarking techniques," *Design automation for embedded systems,* vol. 9, no. 3, p. 211, 2004.

[15] N. Narayan and e. al., "IP Protection for VLSI Designs Via Watermarking of Routes," in *IEEE INternational ASIC/SOC Conference*, Washington, DC, 2001.

[16] 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.