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.

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 $x = 1$.

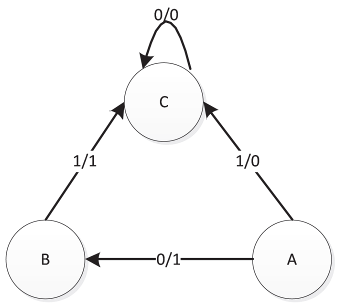
Second, when both flip flops have value 1, the system will be in a state that is not specified in the original FSM. With input $x = 0$ and$x = 1$, the system will output 1 and move to state C and state A, respectively. In other words, the full functionality of the circuit in Figure 1(b) can be described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

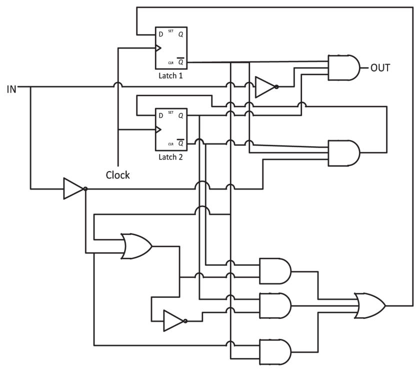
In the original FSM (Figure 1(a)), when the system leaves state A, it cannot come back. In other words, we say that there is no access to state A from state B and state C. However, in the implemented FSM in Figure 1(c), each state has at least one path to come back to state 00. For instance, from state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

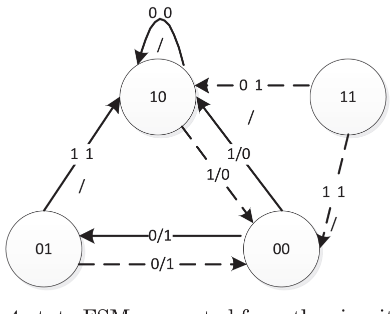
The challenge of building trust into an IC is highlighted in a Defense Science Board study on High Performance Microchip Supply, "Trust cannot be added to integrated circuits after fabrication; electrical testing and reverse engineering cannot be relied on to detect undesired alternations in military 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 $\{s1 \to s2, s2 \to s3, \ldots, sk \to s1\}$ 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

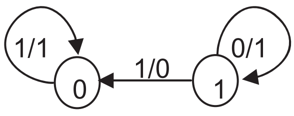


Fig. 2. A simple 2-state FSM.

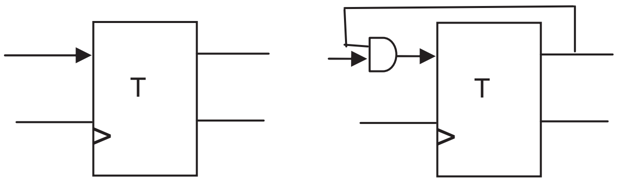


Fig. 3. A normal T flip-flop (on the left) and a T flip-flop with controlled input (on the right).

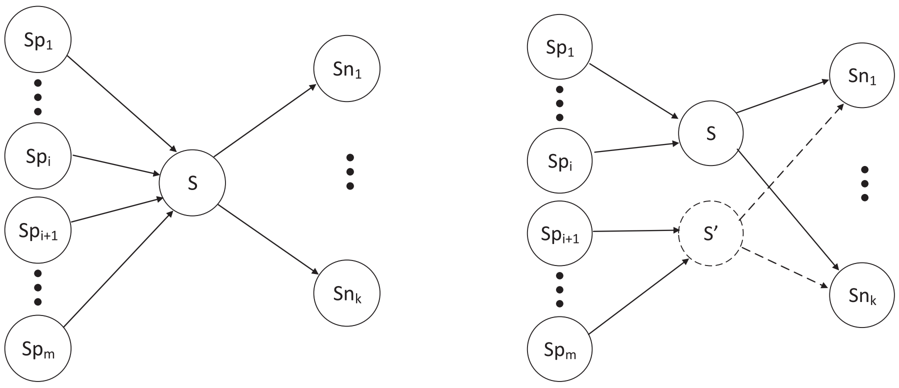


Fig. 4. Reconstructing an FSM by duplicating state S [Yuan et al. 2008].

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 $Sp1, Sp2, \dots, Spm$ and k next states $Sn1, Sn2 \dots, Snk$ (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 countermea-sure
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