Transactions on Design Automation of Electronic Systems

Detection of Un-modeled Combinational Trojans in Full-Scan Designs

Journal:	*Transactions on Design Automation of Electronic Systems*
Manuscript ID:	TODAES-2008-D-409
Manuscript Type:	Paper
Date Submitted by the Author:	30-Apr-2008
Complete List of Authors:	Kocan, Fatih; Southern Methodist University, Computer Science and Engineering Saab, Daniel; Case Western Reserve University, EECS Abraham, Jacob; University of Texas-Austin, ECE
Computing Classification Systems :	combinational Trojan circuits, VLSI , Security, Full-scan design

scholarONE™
Manuscript Central

For Peer Review

Detection of Un-modeled Combinational Trojans in Full-Scan Designs

Fatih Kocan, Daniel G. Saab, and Jacob Abraham, *Fellow, IEEE*.

***Abstract*—The fabrication of Integrated Circuits (IC) is increasingly outsourced for manufacturing. In order to fabricate an IC, design layout details has to be disclosed to the manufacturing facility. This raises the concern of potential caused by manipulation of the hardware and the insertion of extra circuitry (Trojans) into the design. In this paper, we present a process for detecting Trojans in manufactured full-scan ICs. The approach is based on applying scan functions to construct the manufactured circuit connectivity and its embedded functions. The constructed connectivity is compared to the intended design structure to detect Trojans. The functions of the manufactured design are constructed using the discovered connectivity and the scan chain functions. Any Trojan that is not detected during the connectivity check is detected by measuring deviation in the logical behavior of the manufactured functions using function enumeration. We show experimentally that using the proposed approach we were able to detect 100% of randomly injected Trojans in ISCAS89 benchmarks and AES encryption chip.**

***Index Terms*—Trojan Detection, Security, Scan-Design, VLSI Design and Verification.**

I. Introduction

Driven by cheaper fabrication facilities around the world, hardware manufacturers are increasingly outsourcing the fabrication of their ICs. The reduction in the fabrication cost comes at the expense of reduction in security, and the potential loss of intellectual property. This is due to the fact that manufacturers require complete design layout detail for IC fabrication. With access to the design layout details, it is possible that a design can be modified for sinister purposes. These modifications, or Trojans, can cause tremendous damages if not detected by a rigorous process before releasing the manufactured devices. In fact, Trojans are very hard to detect because they are mostly dormant and are triggered by rare events. A triggered Trojan can modify the circuit behavior by enabling extra functions, disabling some functions, or by modifying existing circuit functions. These modifications may cause catastrophic consequences to the ICs and the system that uses them as shown in [17]. For these reasons, hardware Trojans are the cause of great concern in the defense industry [2], [5] as well as in commercial industry. The major cause of this concern is because Trojans are not detected easily by manufacturing test methods. Current test methods do not even consider Trojans. Recently, methods for detecting and protecting against Trojans have been investigated. These include destructive reverse engineering Power/Electromagnetic circuit signature [4], security tagging [3], and unclonable design functions [7]. An analysis of trojan detection based on frequency analysis is preseneted in [6]. Destructive reverse engineering involves stripping all layers of the design and using the extracted layers to reconstruct/re-engineer the manufactured circuit to check for Trojans. This process cannot be performed on every chip and some Trojans that are injected into an untested chip will escape detection. In [4] the authors construct IC fingerprints using power, temperature, and electromagnetic profiles and they show experimentally that large Trojans cause detectable deviation in the profile the make up the IC fingerprints. To make the technique detect small Trojans requires modifications to the IC design process.

Chip integrity and security was not addressed in the original scan design. With trend of implementing encryption algorithms in hardware [20], [18] security is becoming an issue not only for cryptochips but also for general design [13], [21], [23], [19]. In [1], it was shown that scan chain can be used to attack Data Encryption Standard cryptochip by retrieving secret keys and compromising chip security. As a result, techniques have been proposed to address this issue and make the scan chain more secure [11], [14], [15], [16].

In this article, we propose a process and relevant algorithms to detect primarily small combinational Trojans in full-scan circuits. The process can be applied to a secured or unsecured scan design. To our knowledge, it is the *first effective* Trojan detection technique that does not require any modifications to the IC design process, and can be applied to all ICs uniformly. The process precomputes test patterns using an ATPG for every input-gate-output functional dependencies and also test patterns to verify the connectivity of state elements. During detection phase, first, the process applies these dependency verification tests to the manufactured circuit to verify that the dependencies hold as expected. Any broken/added dependency implies detection of a Trojan. If no Trojan is detected in the first step, then the process applies a randomized adaptive dependency verification phase. If no Trojan is detected in the second step as well, then function enumeration step is invoked to detect a Trojan or declare that the circuit is Trojan-free.

The rest of this paper is organized as follows. Section II gives an overview of Trojan circuit concept, full-scan design architecture, and functional dependencies between the nets in a circuit. Section III describes the proposed Trojan detection approaches. To assess the quality of the process, we need to inject Trojans into benchmark circuits. Section IV presents heuristics to inject possible hard-to-detect Trojans. In Section Vthe performance of the proposed process is quantified by

F. Kocan is with Department of Computer Science and Engineering, Southern Methodist University, PO Box 750122 Dallas TX 75275-0122. email: kocan@engr.smu.edu.

D. Saab is with Department of Electrical Engineering and Computer Science, Case Western Reserve University, Cleveland, OH 44106. email: dgs3@cwru.edu.

J. Abraham is with Computer Engineering Research Center, The University of Texas at Austin, Austin, TX 78712. email: jaa@cerc.utexas.edu.

For Peer Review

injecting random Trojans into ISCAS89 benchmarks and AES design. Finally, we draw conclusions and give future directions in Section VI.

II. Background

A. Trojan Circuits

Trojans can either be inserted in the circuit in the form of an extra added circuitry or by removing very small part from the circuit. In both cases, the behavior of the circuit is altered in a way that is hard to detect by inspection, during the testing process, and during normal circuit operation. In these cases, Trojans do not modify directly the primary inputs or outputs in shapes or forms. In fact, Trojans modify the behavior of internal circuit nodes in a way that the behavior of the circuit outputs is the same for most inputs (single or sequence of inputs). The behavior deviation is achieved by a triggering mechanism which wakes up the Trojan. The trigger is a combinations of signals which forms a condition before the trigger begins to alter the circuit behavior and cause errors at intermediate nodes and eventually at the primary circuit output.

Figure 1 shows a Trojan model. In this figure, line L is modified by adding a modification circuit which takes as inputs L and a set of activation signals and generate L^*. L^* is the Trojan signals which replaces L. L^* takes the logical value of L until some logical values are detected on the activation signals. In this case, the logical behavior of L^* will start to deviate from its normal expected causing errors in the circuits. The deviation may start immediately after the combination is detected or may start after some number of clock cycles. We will refer to the first type as combinational Trojans and the second type as sequential Trojans. To make the Trojan hard to detect, the logical combinations of signals must appear rarely during normal circuit operation, and an error on line L must be very difficult to observe on a primary output. Note that L may be more than one line and the activation signals may involve one or more circuit clocks in addition to other nodes. In this paper, we consider combinational Trojans. Other Trojans that involves finite-state machines or those that generate controlled glitches are not the subject of this paper.

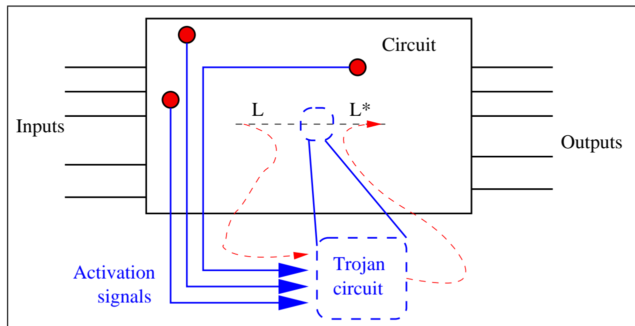

Figure 1. Trojan circuit model

Example 1: Consider c17 in Fig. 2(a), it shows a Trojan free combinational circuit with five inputs, (g_1, g_2, g_3, g_6, g_7), and two outputs (g_{22}, g_{23}). The Trojan free functions are:

$$g_{22} = g_1 \cdot g_3 + g_2 \cdot (\overline{g_3} + \overline{g_6}) \quad (1)$$

$$g_{23} = (g_7 + g_2)(\overline{g_3} + \overline{g_6}) \quad (2)$$

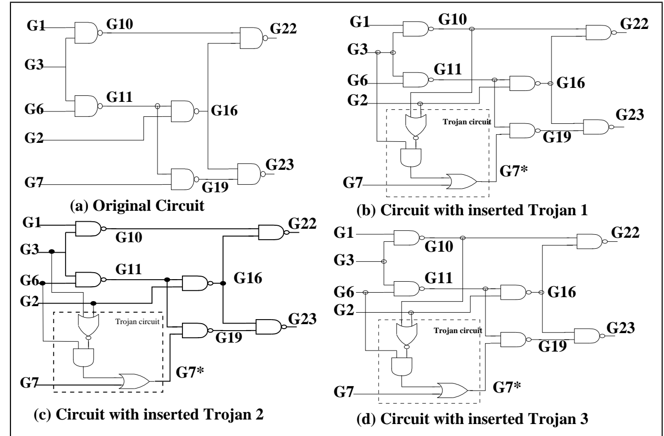

Figure 2. Circuit c17

We inject three Trojan circuits to c17 as in Fig. 2(b), (c), and (d). All these structurally identical three Trojans modify the behavior of line g_7. The table below tabulates g_7^* function and modified g_{23} for these three Trojans.

Fig. 2	g_7^*	g_{23}
(b)	$g_7 + \overline{g_2} \cdot \overline{g_{10}} \cdot g_3$	$g_1 \cdot g_3 \cdot \overline{g_6} + (g_2 + g_7)(\overline{g_3} + \overline{g_6})$
(c)	$g_7 + \overline{g_2} \cdot \overline{g_3} \cdot g_6$	$\overline{g_2} \cdot \overline{g_3} \cdot g_6 + (g_2 + g_7)(\overline{g_3} + \overline{g_6})$
(d)	$g_7 + \overline{g_2} \cdot \overline{g_{10}} \cdot g_6$	$(g_2 + g_7)(\overline{g_3} + \overline{g_6})$

The Trojan in Fig. 2(b) modifies g_{23} and makes it dependent on g_1 in addition to existing dependencies. The one in Fig. 2(c) modifies g_{23} but does not create new dependencies. Finally, the Trojan in Fig. 2(d) is redundant, and would not affect the logical behavior of the circuit but it may affect circuit timing.

B. Full Scan-Based Design

Manufacturing testing of sequential circuits requires generation of test pattern sequences and the application of these sequences to manufactured circuits [12]. Testing of sequential circuits is much complex and time consuming than testing of combinational circuits. This added complexity is due to the state elements (flip-flops) found in sequential circuits. Full-scan-based design methodology allows direct read/write access to the state elements during testing mode of operation, thereby reducing the sequential circuit testing complexity to the combinational circuit testing. The scan architecture requires additional Input/Output Pins and a circuitry which provides a mechanism for the circuit to switch between test and normal modes. During test mode, internal Flip-Flops of the design form into a shift register (i.e., scan-chain register), and the content of each flip-flop in the shift register can be read/written with shift operations.

Figure 3 shows a full-scan sequential circuit. In this figure, the set of primary inputs (outputs) is $\{i_0, i_1, i_2, i_3\}$ ($\{Z_0\}$) while the set of present state lines (next state lines) is

For Peer Review

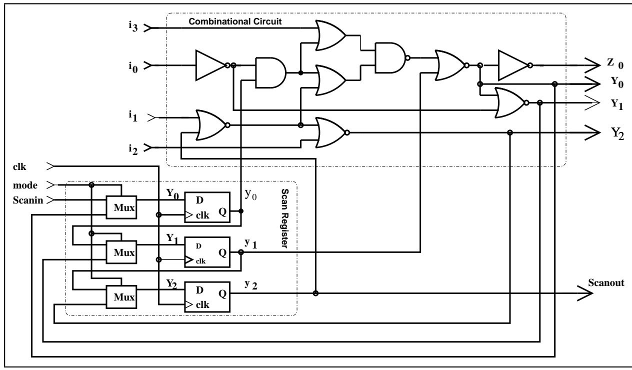

Figure 3. ISCAS89 s27 benchmark circuit.

$\{y_0, y_1, y_3\}$ ($\{Y_0^*, Y_1^*, Y_2^*\}$). The three Flip-Flops forms a shift register. In normal mode ($mode = 0$), $Y_1 = Y_1^*$, $Y_2 = Y_2^*$, and $Y_3 = Y_3^*$. In test mode ($mode = 1$), $Y_0 = scanin$, $Y_1 = y_0$, $Y_2 = y_1$ and $scanout = y_2$. Using the $scanin$ pin, a test vector is scanned into the shift-register by setting $mode$ to 1 and clocking the shift-register using the system clock. After a test vector is fully scanned in, the Flip-Flops are configured to be in normal mode for one clock cycle. At the end of the normal clock cycle, the Flip-Flops loads the normal system response. At this point, the Flip-Flops are again configured into a shift-register to scan-out the stored normal response and concurrently scanning in the next test vector. This process is repeated for all test vectors.

C. Functional Dependencies among the Interconnects

A gate-level design consists of gates and interconnects, i.e., nets. Topology of a design creates structural and/or functional dependencies between the interconnects. A functional dependency implies a structural dependency but a structural dependency may not imply a functional dependency. Multi-level multi-output circuit optimization approaches may create non-functional structural dependencies. Considering a basic gate, the gate's output functionally depends on all of its inputs. Formally, functional dependency is defined as follows.

Definition 1: A function f *functionally* depends on variable y if there exists an input combination such that f is equal to y ($\overline{y}$).

Example 2: In s27 in Fig. 3, the Boolean functions for the primary output (Z_0) and the three pseudo primary outputs (Y_0, Y_1, and Y_2) are:

$$Z_0 = \left(y_2 + \overline{i_3} + i_1\right)\left(\overline{y_0} + i_0\right) + y_1 \tag{3}$$

$$Y_0 = \overline{y_1}\left(\overline{i_1} i_3 \overline{y_2} + \overline{i_0} y_0\right) \tag{4}$$

$$Y_1 = i_0\left(i_1 + \overline{i_3} + y_1 + y_2\right) \tag{5}$$

$$Y_2 = \overline{i_2}\left(i_1 + y_2\right) \tag{6}$$

From these Boolean functions, we build a dependency matrix between the (pseudo) outputs and (pseudo) inputs below. Between an output and an input there can be either functional dependency (f), structural dependency (s) or no dependency ($\cdot$). We note that Y_1 structurally but not functionally depends on y_0.

	i_0	i_1	i_2	i_3	y_0	y_1	y_2
Z_0	f	f	$\cdot$	f	f	f	f
Y_0	f	f	$\cdot$	f	f	f	f
Y_1	f	f	$\cdot$	f	s	f	f
Y_2	$\cdot$	f	f	$\cdot$	$\cdot$	$\cdot$	f

Dependencies provide partial specification of a circuit. Complete specification requires also the conditions when these dependencies must hold. The conditions can be extracted from the Boolean expressions. For example, Y_2 depends on i_1 when $i_2 = 0$, $y_2 = 0$, and other (pseudo) inputs' values are *don't care*, i.e., $Y_2 = \{i_1 0xxxx0\}$. We verify the dependency between i_2 and Y_2 by applying a pair of input patterns that would create a transition at Y_2. Application of $00xxxx0$ and $10xxxx0$ in any order creates a transition: $00xxxx0 \rightarrow 10xxxx0 \Longrightarrow 0 \rightarrow 1$ or $10xxxx0 \rightarrow 00xxxx0 \Longrightarrow 1 \rightarrow 0$.

Definition 2: Assume that b *functionally* depends on a, and let $T_{a \rightarrow b}$ be the complete set of input patterns that establish this dependency. This dependency is *completely* (*partially*) verified iff a transition is observed at b for every pair (some pairs) of input patterns in $T_{a \rightarrow b}$.

We can generalize the dependency analysis and define functional dependencies between the sequences of functionally dependent nets. For example, we can create a functional dependency matrix or a graph for (1) all pairs of nets and outputs, (2) all pairs of nets, or (3) all triples of transtitively functionally dependent nets.

Let $n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow n_3 \ldots \rightarrow n_{k-1} \rightarrow n_k$ be a sequence of structurally dependent nets where n_0 is a primary input and n_k is a primary output. Note that n_i and n_{i+1} do not have to be adjacent nets. This net sequence is functionally dependent iff there exists an input pattern pair that would create a transition on the nets in this sequence. If there is no input pairs creating a transition, then this sequence is a *false sequence*. Note that if the sequence is a path from an input to an output, it would be called *false path*.

Definition 3: A sequence $n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow n_3 \ldots \rightarrow n_{k-1} \rightarrow n_k$ is *completely* (*partially*) verified iff a transition is observed at n_k for every pair (some pairs) of input patterns in $\bigcap_{i=1}^{k-1} T_{i \rightarrow i+1}$.

Example 3: The following dependency matrix shows the functional dependencies between primary inputs and outputs, and all the nets in c17 in Fig. 2(a).

	1	2	3	6	7	10	11	16	19
1	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	f	$\cdot$	$\cdot$	$\cdot$
2	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	f	$\cdot$
3	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	f	f	f	f
6	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	f	f	f
7	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	f
22	f	f	f	f	$\cdot$	f	f	f	$\cdot$
23	$\cdot$	f	f	f	f	$\cdot$	f	f	f

$$g_{22} = g_2 \cdot (\overline{g_3} + \overline{g_6}) + g_1 \cdot g_3 \tag{7}$$

$$g_{16} = \overline{g_2} + g_3 \cdot g_6 \tag{8}$$

$$g_{22} = \overline{g_{16}} + g_1 \cdot g_3 \tag{9}$$

$$g_{23} = \overline{g_{16}} + g_7 \cdot \overline{g_3 \cdot g_6} \tag{10}$$

For Peer Review

We can verify the dependencies between all possible input-net-output triples. As an example, to verify the functional dependency of $g_3 \rightarrow g_{16} \rightarrow g_{22}$, we need to apply a pair of test patterns that would create a transition at g_3, pass through g_{16} and be observed at g_{22}. The test set to create such a transition is $T_{g_3 \rightarrow g_{16} \rightarrow g_{22}} = T_{g_3 \rightarrow g_{16}} \cap T_{g_{16} \rightarrow g_{22}} = \{01g_3 1x\}$, where $T_{g_3 \rightarrow g_{16}} = \{x1g_3 1x\}$, and $T_{g_{16} \rightarrow g_{22}} = \{0xxxx, xx0xx\}$.

III. Trojan Detection Approaches

In this section, we presents methods to verify the functional correctness of a manufactured design. We assume that a description of the golden design (the design which is to be fabricated) is available, which consists of the input/output as well as the internal design detail. In addition, we have a manufactured design in which only the input/output behavior is available and it may be a Trojan-contaminated design. With these assumptions, our approach consists of three heuristics and one deterministic steps: (1) scan-register and state verification, (2) design and manufacturing test verification, (3) functional dependency verification, (4) random verification, and (5) complete functional verification. Any one of these steps may detect the Trojan, if there is one, and terminate the verification process. In this work, we develop techniques and algorithms for steps 1, 3 and 4 since the test sets for step 2 would be generated with existing algorithms and step 5 is merely exhaustive function enumeration. In order to declare that the manufactured IC is Trojan free, we must enumerate all min-terms of the design. Note that steps $2-5$ must be tried after step 1.

A. Verification of State Elements (Flip-Flops) and Scan Circuitry

The scan chain consists of a shift register where each flip-flop corresponds to a state variables. The flip-flop can store a logic '0' and a logic '1' and the combination stored in the scan chain can be shifted and observed one each clock cycle at the scan out. To compute the number of flip-flops are in the chain, we shift a logic '1' followed by a large number n of '0' (n is larger than the number of flip-flops in the chain). Concurrently while shifting into the scan, we observe the scan out and record the clock cycle when the last '1' is observed at the scan-out line. We are shifting one '1' followed by n zeros into the scan chain. Thus, at clock cycle K in which the last '1' appears indicates the size of the scan chain. Each flip-flop is given a identification number between 1 and K. These numbers are used to identify variables corresponding to present-states y_i's and next-states Y_i's.

Consider the circuit shown in Figure 3 and assume the initial state of flip-flops ($y_0,\ y_1,\ y_2$) is '(101)' as shown in the table below. A sequence of 0000001 is serially shifted into the scan one bit/clock. The sequence observed at the scan-out at each clock is 1011000. During the 3th clock, the last logic '1' is observed at the scan out line. This indicates that the length of the scan chain for this circuit is 3 flip-flops. If a Trojan changes the scan-chain in a way that increases or decreases the number of state elements, then it will be detected. If the scan-chain is re-arranged, then it may not be detected in this step but it will be detected in the steps that follow.

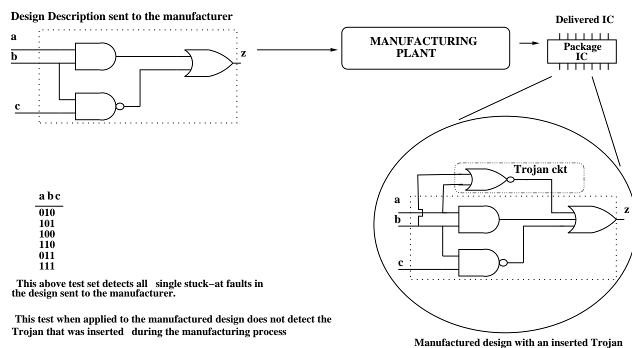

Figure 4. A Trojan that escapes stuck-at tests

scanin	y_2	y_1	y_0	scanout	clockcycle
1	1	0	1	1	0
0	1	1	0	0	1
0	0	1	1	1	2
0	0	0	1	1	3
0	0	0	0	0	4
0	0	0	0	0	5
0	0	0	0	0	6

B. Verification of Manufacturing and Design Tests

Test sets are generated to verify a design and also to detect manufacturing defects in a manufactured circuit. We can use these test sets to detect a Trojan as follows. We simulate all the test patterns and store the correct responses. During Trojan detection, we apply these tests to the manufactured circuit and compare produced responses with the stored responses. If any mismatch occurs, the circuit is either defective or Trojan contaminated. In any case, the chip is discarded.

Trojans may escape manufacturing and design verification tests (step 2) such as stuck-at test, delay test set etc. Figure 4 shows a design which is to be sent to a manufacturing plant. In fact, these test, at best, they can detect deleted dependency but can not reveal added Trojan dependencies. The test set which detects all stuck-at faults in the sent design is shown in the table below. During the manufacturing process, a simple Trojan circuit, as shown, is inserted in the design and delivered as part of the the final packaged IC design. Note, this Trojan is not detected by the test set. The activation signal values (00) on (ab) are not part of the test set. This Trojan will escape the testing process. In fact, for every test set one can find a Trojan circuit that can be inserted in the design that the test will not activate. This is especially, easy to achieve, when minimal test sets are used. In general, test methods do not do exhaustively enumerate all possible input combinations. That is why a Trojan that is activated by a pattern that is not part of the test sequence will escape detection. In fact, for every test pattern there is a set of Trojans that will escape the testing process. This is why one needs a new methods that aim at detecting Trojans.

For Peer Review

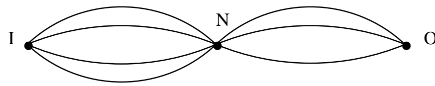

Figure 5. Paths from input I to output O via net N

C. Verification of Functional Dependencies

In an effort to detect a Trojan before costly function enumeration step, we develop a detection method based on verification of functional dependencies between the nets in a gate-level design. We generate test set to verify if functional dependencies hold in the manufactured circuit. Algorithm 1 finds a test pattern pair to verify functional dependency between every primary input, net, and primary output triples. The algorithm can be generalized to find functional dependency test set for any sequence of nets. Nevertheless, as we increase the sequence length, this process may reduce to full function enumeration. Therefore, we limit our sequences to triples.

Algorithm 1 proceeds as follows. First, for each gate g, it finds the set of primary inputs on which g functionally depends ($ID(g)$), and the set of primary outputs that functionally depend on g ($OD(g)$). Second, for each input, gate and output triples, it calls ATPG to find a test pair to be used for dependency verification. A test pair may capture additional dependencies. Last, the algorithm adds the newly detected triples to the set of detected triples.

The method uses ATPG techniques to find a pair of test patterns to propagate a transition. The problem of finding a transition test pair can be reduced to the finding a test for multiple-stuck-at faults. We use Fig. 5 to illustrate ATPG-based test pair finding. Let I, N, and O be an input, a net, and an output, respectively. To propagate a transition from I to O via N, we assume that there is a combination of stuck-at-0/1 faults at nets I, N, and O that can be detected by a pair of tests. The number of stuck-at faults combinations is 8. Some of these combinations of faults may not be detectable by a pair of tests.

Example 4: The input set and the output set of g_{16} are $ID(g_{16}) = \{g_2, g_3, g_6\}$ and $OD(g_{16}) = \{g_{22}, g_{23}\}$, respectively. From example 3, the test set that reveals the functional dependency among g_3, g_{16}, and g_{22} is $\{01g_31x\}$. Assume that ATPG returns $\{01010, 01111\}$ as the two test patterns. Application of $01010 \rightarrow 01111$ would create a transition along two paths: (1) $g_3 \rightarrow g_{11} \rightarrow g_{16} \rightarrow g_{23}$ and (2) $g_3 \rightarrow g_{11} \rightarrow g_{16} \rightarrow g_{22}$. The test pair detects the dependencies among g_3, g_{16} and g_{22} also detects $\{g_3, g_{11}, g_{22}\}$, $\{g_3, g_{11}, g_{23}\}$, and $\{g_3, g_{16}, g_{23}\}$. Therefore, we mark them as detected and don't invoke ATPG to find test pairs for them.

D. Randomized Adaptive Verification

In this section, we devise a randomized adaptive verification algorithm to detect a Trojan. The randomized algorithm works as follows. It applies a random pattern (t_0) to initialize the circuit under Trojan detection. After that, it searches for a random pattern (t_i, $1 \leq i \leq 4$) such that application of $t_0 \rightarrow t_i$ would cause a transition in at least 10% of the primary outputs. If none of the four pairs in $\{t_0 \rightarrow t_1,\ t_0 \rightarrow t_2\ t_0 \rightarrow t_3,\ t_0 \rightarrow t_4\}$

Algorithm 1 Dependency identification and test set generation

Require: Gate-level circuit G
  $ID(i) \leftarrow \{i\}$ for every input i
  for $l = 2;\ l \leq maxlevel;\ l++$ **do**
    for each gate g at level l **do**
      $ID(g) \leftarrow \bigcup_f\ ID(f),\ f \in Fanin(g)$
    end for
  end for
  $OD(i) \leftarrow \{i\}$ for every output i
  for $l = maxlevel - 1;\ l \geq 1;\ l--$ **do**
    for each gate g at level l **do**
      $OD(g) \leftarrow \bigcup_f\ OD(f),\ f \in Fanout(g)$
    end for
  end for
  for each gate g $g \in G$ **do**
    for each input $i \in ID(g)$ **do**
      for each ouput $o \in OD(g)$ **do**
        if $\{i, g, o\} \notin Detected$ **then**
          find a pair of input patterns using ATPG
          compute the set of detected dependencies (DD)
          $Detected \leftarrow Detected \cup DD$
        end if
      end for
    end for
  end for

Algorithm 2 Dependency verification with a test set

  for each (input i, net n, output o) triples **do**
    for each (t_i, t_j) test pair in TestSet(i, n, o) **do**
      $POs(t_i) \leftarrow apply_input(t_i)$
      $POs(t_j) \leftarrow apply_input(t_j)$
      if $\exists$ a transition at output m and $i \notin ID(F_m)$ **then**
        return Trojan Detected
      end if
      if No transition at output o **then**
        return Trojan Detected
      end if
    end for
  end for

causes at least 10% transitions at the primary outputs, then test t_i causing the maximum number of transitions is selected. After the selection of t_i, the primary input values in t_i are flipped one by one to derive new patterns. Let $t_{i,k}$ be the derived pattern obtained by flipping the kth input in t_i. Each derived $t_{i,k}$ is applied to the circuit one after another: $t_i \rightarrow t_{i,0} \rightarrow t_{i,1} \rightarrow t_{i,2} \rightarrow \ldots \rightarrow t_{i,n-1}$, where n is the number of inputs. If any two consecutive pair of test patterns deterministically derived from t_i creates a transition in an output (i.e., a dependency is detected), then we gain from t_i therefore we set $gain$ to $TRUE$ to reward the search. Otherwise, $gain$ is false for the random pattern t_i. Moreover, the algorithm detects a Trojan if upon changing the kth input we observe a transition at output o_j and o_j does not functionally depend on input k in the golden design ($D(F_n) \neq ID(F_n)$). At the end of flipping all input values, if

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

$gain$ is set to $TRUE$, the algorithm continues flipping input bits of the last pattern ($t_{i,n-1}$) starting from the first input.

Algorithm 3 Randomized adaptive dependency verification

Require: Iter_Limit
  $iter \leftarrow 0$
  $t_0 \leftarrow Generate_Random_Pattern()$
  while $iter < Iter_Limit$ **do**
    for $i \leftarrow 1$; $i < 5$; $i \leftarrow i + 1$ **do**
      $t_i \leftarrow Generate_Random_Input()$
      $POs(t_0) \leftarrow apply_input(t_0)$
      $POs(t_i) \leftarrow apply_input(t_i)$
      $diff(i) \leftarrow$ Number of transitions at POs
    end for
    t is the t_i that yields maximum $diff$.
    $gain \leftarrow TRUE$; $Ggain \leftarrow FALSE$
    while $gain$ **do**
      $gain \leftarrow FALSE$
      $POs(t) \leftarrow apply_input(t)$
      for $k \leftarrow 0$; $k < |PIs|$; $k++$ **do**
        $t \leftarrow flipbit(t, k)$
        $POs(t_k) \leftarrow apply_input(t_k)$
        $diff \leftarrow$ Number of transitions at POs
        if $diff > 0$ **then**
          $gain \leftarrow TRUE$
        end if
        if $\exists$ a transitioning output n s.t. $k \notin ID(F_n)$ **then**
          Return Trojan Detected
        end if
      end for
      if $gain$ **then**
        $Ggain \leftarrow TRUE$
      end if
    end while
    if $Ggain$ **then**
      $iter \leftarrow 0$
    else
      $iter \leftarrow iter + 1$
    end if
  end while
  Return No Trojan Detected;

If the last random pattern t_i sets $gain$ to $TRUE$, the Global gain ($Ggain$) flag is set to $TRUE$ as well. If during a single iteration no dependency is detected, then the search is terminated and a new random test pattern is generated. The random test generation is repeated *Iter_Limit* and is rewarded if any $gain$ is recorded in the previous iteration. The process terminates and return no detected Trojans when *Iter_Limit* consecutively generated random patterns does not cause any $gain$. In this work, *Iter_Limit* is set to 10000.

We conclude this section with a remark. An alternative random verification approach would be the following. After generation of a random pattern, we can apply this random pattern to the manufactured circuit and also to the software-model of the golden design. After that, we can simply compare the logic values computed by the simulation and the chip. If there is any mismatch, we detect the Trojan. However, this alternative implementation requires costly software simulation algorithm for every random test pattern. Our procedure,3, does not require the simulation of the software model for every test pattern. Our proposed approach detects Trojans using mismatch in the precomputed functional input-output dependencies.

E. Function Enumeration

At this point, we have checked that, for every function, its dependent design inputs are implemented. This does not guarantee that Trojans are not in the design. In fact, the design may still contain Trojans that cause some manufactured functions behavior to deviate from the behavior of the designed ones. To detect these type of Trojans, we have to compute the manufactured functions. The function computation is a process that enumerates the input combinations of the function supporting variables and implying those combinations in the manufactured circuit, using the scan functions, to deduce the logical behavior of the output functions. This process is computationally expensive and can only be performed on function with a limited number of inputs. In our case, we generated complete logical behavior for circuits which have functions with less than 32 inputs. In addition, those functions were generated and compared to the designed ones and were not stored. For functions with large number of supporting inputs, the combinations are randomly generated in way that increases the likelihood of Trojan detection.

IV. Trojan Insertion Method for Experimentation

We assess the quality of our proposed Trojan detection approaches by injecting random combinational Trojans into the benchmark circuits and reporting the detection ratio. Instead of pure random injection, we use some heuristics that would make a Trojan hard-to-detect. The heuristics guide the selection of Trojan inputs (*Trojan activation signals*) and Trojan output injection line (*Trojan signal*).

A. Activation Signal Selection

A d-input Trojan circuit $f(n_1, n_2, \ldots, n_d)$ must escape the majority of functional patterns so that it becomes a hard-to-detect circuit. Thus, there should be only a small number of functional patterns that would rarely activate the Trojan circuit and cause erroneous outputs due to its activation. A Trojan circuit is hard-to-detect if (1) it rarely modifies the behavior of the Trojan signal (i.e., injected line) and (2) the modified behavior is rarely observed at any outputs. Assuming that a Trojan circuit f is active (inactive) when $f = 1$ ($f = 0$), rare activation of f occurs if (i) there is only one input combination that sets f to 1 and (ii) this particular input combination occurs rarely. In order to find the rarely occurring input combination, we simulate the Trojan-to-be-injected circuit with a set of (possibly randomly generated) input patterns and count the frequency of every input combination on some selected set of d-tuple nets separately. The lowest frequency input combination at a d-tuple net set indicates that this

For Peer Review

input combination is a rarely occurring candidate combination (for the simulated patterns) and its respective nets should be connected to the inputs of the Trojan circuit. Hence, the rare input combination is the only min-term of the Trojan circuit. In addition to frequency of input combinations at d-tuple nets, we need to keep track of the patterns that set these combinations. The patterns sets would be used in the selection of a Trojan signal. The proposed Trojan signal selection method follows a formalization of the aforementioned activation signal selection method.

Let $v_1 \dots v_d$ be an input combination, $g_1, \dots, g_d$ be a d-tuple net set, $f^{v_1 v_2 \dots v_d}_{g_1, g_2, \dots, g_d}$ be the frequency of observing $g_1 = v_1 \wedge g_2 = v_2 \wedge \dots \wedge g_d = v_d$ where g_i is the ith net in the target circuit and v_i is its logic value, and $T\left(g_1 = v_1\,, \dots,\; g_d = v_d\right)$ be the set of patterns that sets $g = v_1 \wedge \dots \wedge g_d = v_d$. Also, let $F(s) = \left\{ \{g_i = v_1, g_j = v_2, \dots, g_k = v_d\} \mid f^{v_1 v_2 \dots v_d}_{g_i, g_j, \dots, g_k} = s \right\}$. We explain this notation with an example below.

Example 5: Exhaustive simulation c17 in Fig. 6 results in the following lowest frequencies for $d = 2$: $f^{00}_{10,16} = 2$, $f^{00}_{10,19} = 2$, $f^{10}_{2,22} = 2$ and $f^{01}_{10,23} = 3$. Thus, $F(2) = \{\{g_{10} = 0, g_{16} = 0\}, \{g_{10} = 0, g_{19} = 0\}, \{g_2 = 1, g_{22} = 0\}\}$ and $F(3) = \{\{g_{10} = 0, g_{23} = 1\}\}$. Also, $T\left(g_{10} = 0, g_{16} = 0\right) = \{1110x\}$, $T\left(g_{10} = 0, g_{19} = 0\right) = \{1x101\}$, $T\left(g_2 = 1, g_{22} = 0\right) = \{0111x\}$, and $T\left(g_{10} = 0, g_{23} = 1\right) = \{1110x,\, 1x101\}$. Based on these results, we pick one element from $F(2)$ as the set of activation signals for the Trojan circuit. However, the decision is not final and in some situations we may need to change our decision and try another activation signal set. We discuss this in Section IV-B.

It is impractical to compute the frequencies for every possible activation signal set, especially for the large values of d. A circuit with N nets requires an update of $2^d \cdot \binom{N}{d}$ entries per a simulated pattern with a straightforward algorithm. We can avoid this costly computation as follows. First, we simulate the circuit for all patterns to collect frequencies of all nets for $d = 1$. We identify the K lowest frequency signal set, say $K \leq 100$. Second, we re-simulate the circuit with all patterns and collect frequencies of activation signal sets that involve only the selected K nets.

B. Trojan Signal Selection

After computing activation signal sets, we identify potential nets to inject the Trojan. The nets with high observability values indicate that it is hard to propagate the logic value on this net to any outputs. Hard propagation implies rare detection of a rarely activated Trojan circuit and rare propagation of Trojan effect. The Trojan circuit may modify the behavior of an injected net L in three ways as shown in Table I. Each Trojan effect is valid provided that the condition is satisfied. In this table, $T(e)$ is the set of functional patterns that activate the Trojan and $T(L = i)$ is the set of patterns that set L to i. The conditions guarantees that there is at least one pattern that would activate the Trojan and cause inversion of the logic value at L. In case, the selection of activation signal set e does not satisfy the condition for the desired effect, we can either pick the next lowest frequency signal set or change the Trojan signal to a net with the next largest observability value.

Table I
TROJAN'S EFFECT

Trojan (τ)	L	Effect	Condition	Implementation
0	x	$L^* = L$	-	-
1	x	$L^* = 1$	$T(e) \cap T(L = 0) \neq \phi$	$L^* = \tau \vee L$
1	x	$L^* = 0$	$T(e) \cap T(L = 1) \neq \phi$	$L^* = \overline{\tau} \wedge L$
1	x	$L^* = \overline{L}$	$T(e) \neq \phi$	$L^* = \tau \oplus L$

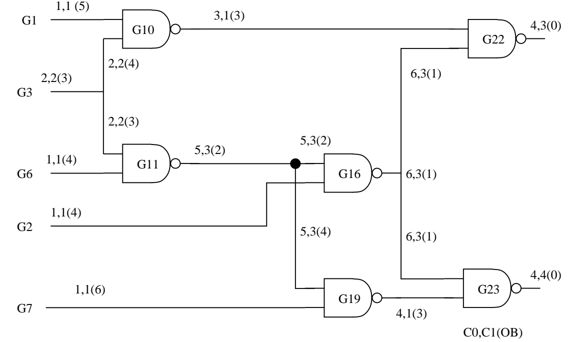

Figure 6. Observability and controllability values for c17

Example 6: We compute 0- and 1-controllability ($C0$, $C1$) and observability (OB) values for c17 in Fig. 6. We compute these testability measures using Goldstein's method [24] with two minor modifications: (1) We add the number of fan-outs at a stem to both of its controllability values if the number of fan-outs is at least two, and (2) Cost of a gate is assumed to be zero in the calculation of controllability and observability. With this method, we found that G7 has the highest observability value, and $T(g_7 = 1) = \{xxxx1\}$ and $T(g_7 = 0) = \{xxxx0\}$. For activation signal set $e = \{g_{10} = 0, g_{16} = 0\}$, $T(e) \cap T(g_7 = 1) = \{1110x\} \cap \{xxxx1\} = \{11101\}$, $T(e) \cap T(g_7 = 0) = \{1110x\} \cap \{xxxx0\} = \{11100\}$. However, for $L = g_1$ $e = \{g_2 = 1, g_{22} = 0\}$, $T(e) \cap T(g_1 = 1) = \{0111x\} \cap \{1xxxx\} = \phi$. Therefore, for this case, it is impossible to achieve $L^* = 0$ effect since the corresponding condition does not hold.

V. Experimental Results

The procedure outlined in the previous sections was implemented in a computer program. The input data to the program is in the form of a Verilog file. We evaluated the procedure using some of the ISCAS'89 benchmark circuits [9] and an implementation of an AES encryption chip [10].

The first step in the process is the state verifications. In this step, the algorithm verify that the manufactured circuit has the same number of states in the scan chain as in the original design. Table II shows the state verification results. Each row in this table shows the number of Flip-Flops (FF), the number of gates (Gates), the number of input (In), and the number of output (Out) in the original design. The rest of the row shows the number of shifts applied (# Shifts) to the manufactured design and the number of state found identified in the manufactured design. As a result of this procedure, the position of each state identified in the scan register is revealed.

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

Table II
STATE VERIFICATION

Circuit	FF	Gates	In	Out	Shifts	y_i
s298	14	119	3	6	3000	14
s344	15	160	9	11	3000	15
s349	15	161	9	11	3000	15
s382	21	158	3	6	3000	21
s386	6	159	7	7	3000	6
s400	21	163	4	6	3000	21
s420	16	218	18	1	3000	16
s444	21	181	3	6	3000	21
s510	6	211	19	7	3000	6
s526n	21	194	3	6	3000	21
s526	21	193	3	6	3000	21
s641	19	379	35	24	3000	19
s713	19	393	35	23	3000	19
s820	5	289	18	19	3000	5
s832	5	287	18	19	3000	5
s953	29	395	16	23	3000	29
s1196	18	529	14	14	3000	18
s1238	18	508	14	14	3000	18
s1488	6	653	8	19	3000	6
s1494	6	647	8	19	3000	6
s5378	179	2779	35	49	3000	179
s9234	211	5597	36	39	3000	211
s35932	1728	16065	35	320	3000	1728
AES	533	68805	261	130	3000	533

We performed 3000 shifts for each circuit. In general, we may want to perform more shift to reduce uncertainties in the number and position of state variables.

The second step in the process is the verification of functional dependencies. For this, we use Alg. 2 to identify the functional dependencies and generate test pairs for them. Table III shows the total number of dependencies for ISCAS'89 benchmarks and AES circuit. The table also reports the CPU time needed for this computation. In this table, we see that s298 contains 86 structural dependencies and 83 functional ones. This means that we generated a total of 83 pairs of tests one for each dependency. The CPU time needed to compute the information for the AES is large. In this circuit, there is a large number of hard to test faults. Each of these faults is targeted by ATPG and no fault simulation was performed. In addition, we have to supply a very large backtrack limit because we do not want any fault to be dropped. A Dropped fault causes ambiguity in the functional dependency generation which makes the Trojan detection procedure unreliable.

Table III shows the result of running the detection process that includes application of dependency verification test set (Algo. 2) and randomized adaptive verification (Algo. 3). To evaluate the effectiveness of this process, we inject in each circuit a total of 16 Trojans using the approach in Section *IV*. Thus, we run the Trojan detection process 16 times for each circuit. In this experiment, an inserted Trojan is small and consists of no more than two activation signals. As a result, the size of the injected Trojans is no more than three gates. The pairs of signals, forming the activation condition, are chosen in a way that the condition rarely appears. The rare conditions are determine by simulating each circuit by large number of random inputs (200000) and the pairs of patterns that appears the least on corresponding nodes are chosen. Based on this patterns, the function of Trojan is decided. A line with low observability [12] is selected to be modified by the condition. The table III shows the number of Trojans detected during the dependency verification process and during the randomized adaptive process. The CPU time required by the dependency verification process is much less than the randomized process. For this reason, the detection of Trojans in circuits consisting of functions with sparse dependencies is most likely accomplished in the latter process. For circuit with dense input-output dependencies, Trojans are most likely detected by the former process. Also, we should note that the CPU time is for running the process 16 times. This means that for a single run, the program requires $\frac{1}{16}$ of the reported CPU time.

The Trojans undetected by the above procedure can only

Table III
FUNCTIONAL DEPENDENCIES

ckt	Dependency Matrix Structural	Functional	CPU
s298	86	83	.003s
s344	121	121	.007s
s349	121	121	.005s
s382	175	172	.008s
s386	129	129	.009s
s400	175	172	.008s
s420	186	186	.014s
s444	175	172	.010s
s510	103	103	.010s
s526	167	164	.009s
s526n	167	164	.010s
s641	500	500	.054s
s713	486	486	.087s
s820	213	213	.025s
s832	213	213	.027s
s1196	387	375	.070s
s1238	387	375	.067s
s1488	266	266	.068s
s1494	266	266	.068s
s5379	2313	2233	4.031s
s9234	3260	2861	41.181
s35932	7595	7339	1m23
AES	7860	7860	5h20m44

Table IV
DETECTION USING $DetectionProcess$

ckt	Detected Assert	Generate	Undetected	CPU
s298	6	9	1	128.01s
s344	8	6	1	1.52s
s349	4	11	2	2.43s
s382	7	8	1	204.80s
s386	10	4	2	2.62s
s400	12	2	2	203.55s
s420	10	6	0	277.76s
s444	15	1	0	232.80s
s510	3	13	0	2.52s
s526	7	8	1	240.96s
s526n	7	8	1	241.68s
s641	5	11	0	698.56s
s713	6	6	4	697.56s
s820	3	13	0	123.77s
s832	4	12	0	300.32s
s1196	1	15	0	18.07s
s1238	2	12	2	21.66s
s1488	1	15	0	24.96s
s1494	2	14	0	25.63s
s5379	3	11	2	12219.84s
s9234	1	13	2	28135.50
s35932	14	1	1	3d4h
AES	5	6	5	7d7h

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

Table V
DETECTION USING FUNCTION ENUMERATION

ckt	time (sec)	Num of test	Detected
s298	4.56	15250	1
s713	8.216	102041	4
s1238	14.760	345000	2
s5378	118.616	108650	2
s9234	286.000	125990	2
s35932	1918.680	250000	1
AES	22d3h	2^{25}	5

be detected by generating the truth table of the functions implemented in the manufactured circuits. For the small circuits, we were able to detect all the undetected Trojans using exhaustive enumeration. We should mention that once a Trojan is detected, the input enumeration stops. In fact, in all cases the enumeration stopped but did not enumerate all 2^n combinations (n number of inputs on which the function depends). The results are shown in Table V. The AES circuit, the number of test tried, before detecting the Trojans, is 2^{25}.

For large circuits, we noticed that in every circuit most of the functions are small with few exceptions. The profile of the functions with less than 31 inputs is shown in Figure 7 and the profile of circuit functions with more than 31 inputs is shown in Figure V. In these charts, the number of inputs is shown on the x-axis and the number of functions with that many inputs is plotted on the y-axis. Note, that the majority of the functions in these circuits have less than 31 inputs and much fewer functions have more than 31 inputs. For these function, we used pseudo-random inputs to detect Trojans. In our experiment, we were able to detect, using the function generation, all Trojans that were not detected by the dependency verification test and randomized processes. We observed that Trojans that infect small functions can be easily and definitely detected while Trojans that infect large functions are much harder to detect. This is due to the large number of combinations that needs to be applied in order to exhaust all input combinations. Therefore, a circuit may be Trojan resistant if it is designed using functions that depend on small number of inputs which can be achieved by partitioning large functions and by pipe-lining.

VI. CONCLUSION

In this paper, we presented a technique for detecting Trojans in manufactured full-scan integrated circuits. The approach is based on discovering circuit connectivity and on computing part of the implemented functions. The approach discovers/constructs the implemented connectivity to detect Trojans that cause connectivity deviation. The connectivity knowledge is used to discover the logic of the implemented functions. Any Trojan that is not detected during the connectivity check is detected by measuring deviation in the logical circuit behavior. We show experimentally that using the proposed approach we were able to detect 100 % of the injected Trojans.

REFERENCES

[1] B. Yang, K. Wu, and R. Karri, Scan Based Side Channel Attack on Dedicated Hardware Implementations of Data Encryption Standard, in Proc. IEEE ITC, 2004, pp. 339-344.

[2] Defense Science Board Task Force. High performance microchip supply, *http://www.acq.osd.mil/dsb/reports/2005-02-HPMSReportFinal.pdf*. Feburary 2005.

[3] Carol Marsh, "A Security Tagging Scheme for ASIC Designs and Intellectual Property Cores" newblock *http://www.us.design-reuse.com/articles/article15105.html*. 2007.

[4] D. Agrawal, S. Baktir, D. Karakoyunlu, and P. Rohatgi, and B. Sunar. "Trojan Detection Using IC Fingerprinting", in Proc. IEEE Symp. on Security and Privacy, May 20-23, 2007.

[5] Trust in Integrated Circuit (TIC), *http://www.darpa.mil/MTO/solicitations/baa07-24/index.html*. Feburary 07.

[6] Francis Wolff, Christos Papachristou, Rajat Subhra Chakraborty and Swarup Bhunia, "Towards Trojan-Free Trusted ICs: Problem Analysis and a Low-Overhead Detection Scheme," Design Automation and Test in Europe (DATE), Mar 10-14, 2008.

[7] G. E. Suh and S. Devadas. "Physical unclonable functions for device authentication and secret key generation", Proceedings of the Design Automation Conference (DAC), 2007.

[8] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits," Proc. IEEE ISCAS, 1985, pp. 695-698.

[9] F. Brglez, D. Bryan, and K. Kozminski. "Combinational Profiles of Sequential Benchmark Circuits," IEEE ISCAS, May 1989.

[10] R. Usselmann. aes cipher ip core. In *http://www.opencores.org*, 2002.

[11] B. Yang, K. Wu, R. Karri. Secure Scan: A Design-for-Test Architecture for Crypto Chips, IEEE Trans. on CAD, 2006, pp. 2287-2293.

[12] M. Abramovici, Melvin A. Breuer, , and Arthur D. Friedman. Digital systems testing and testable design. *CS Press*, 1990.

[13] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Ravi, Security as a New Dimension in Embedded System Design, in DAC, June 2004, pp. 753-760.

[14] D. Hely, et al. Scan Design and Secure Chip, in Proc. of the 10th IEEE Intl. On-Line Testing Symposium, 2004.

[15] J. Lee, M. Tehranipoor, C. Patelm and J. Plusquellic, Securing Scan Design Using Lock and Key Technique, in Proc. of IEEE Symp. on DFT, 2005.

[16] J. Lee, M. Teharanipoor, and J. Plusquellic, A Low-Cost Solution for Protecting IPs Against Scan-Based Side-Channel Attacks , VTS 2006.

[17] D. Boneh, R. DeMillo, and R. Lipton, "On the importance of checking cryptographic protocols for faults," In EUROCRYPT '97, vol. 1233 of Lecture Notes in CS, pp. 37-51. Springer-Verlag, 1997.

[18] K. Hafner, et al., Design and Test of an Integrated Cryptochip, IEEE D&T, pp. 6-17, Dec. 1991.

[19] S. Mangard, M. Aigner, and S.Dominikus, A highly regular and scalable AES hardware architecture, IEEE Trans. Comput., vol. 52, no. 1, pp. 483-491, Apr. 2003.

[20] I. Verbauwhede, P. Schaumont and K. Kuo, Design and performance testing of a 2.29 Gb/s Rijndael processor, in IEEE Journal of Solid-State Circuits, Mar. 2003, pp. 569-572.

[21] K. Tiri and I. Verbauwhede, A VLSI Design Flow for Secure Side-Channel Attack Resistant ICs, in Proc. of DATE, Mar. 2005, pp. 58-63.

[22] R. Karri, K. Wu, and P. Mishra, Fault-Based Side-Channel Cryptanalysis Tolerant Architecture for Rijndael Symmetric Block Cipher, in IEEE Intl. Symp. on DFT, 2001, pp. 427-435.

[23] S. Ravi, A. Raghunathan, and S. Chakradhar, Tamper Resistance Mechanisms for Secure Embedded Systems, in Proc. of the 17th Intl. Conf. on VLSI Design, 2004, pp. 605-611.

[24] Niraj K. Jha and Sandeep Gupta, Testing of Digital Systems, Cambridge University Press, ISBN: 0521773563, 2002.

For Peer Review

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

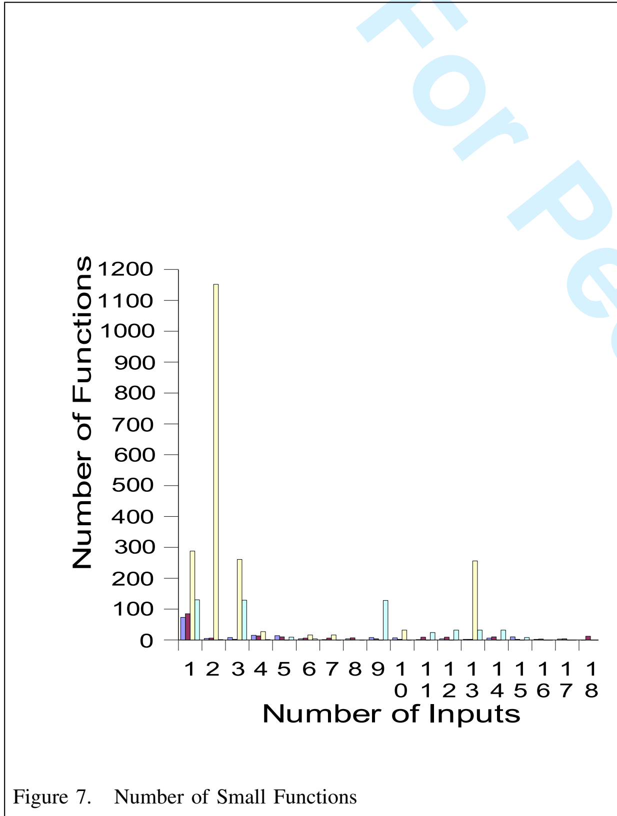

Figure 7. Number of Small Functions

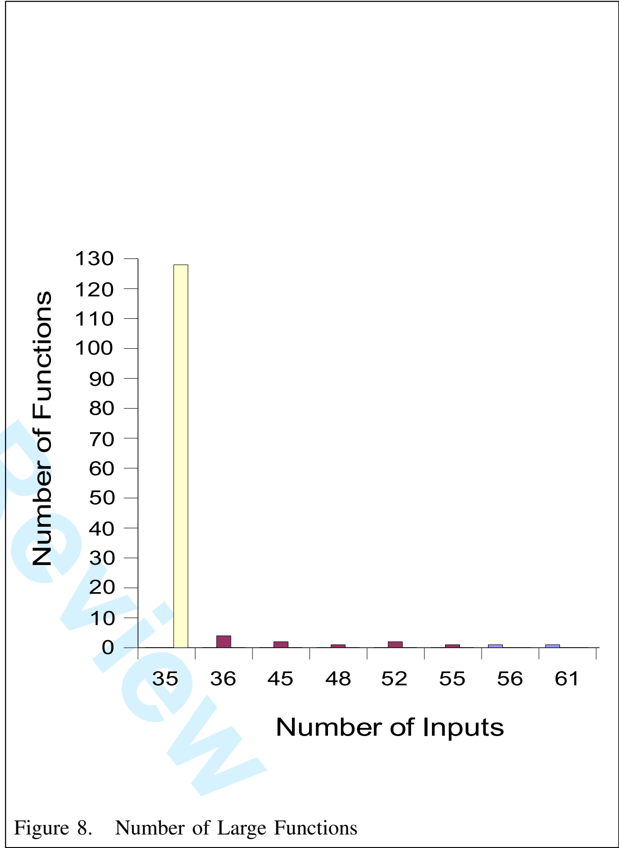

Figure 8. Number of Large Functions

For Peer Review