Security Analysis of Integrated Circuit Camouflaging

Jeyavijayan Rajendran
ECE Department
Polytechnic Institute of NYU
Brooklyn, NY, USA, 1120
jv.ece@nyu.edu

Michael Sam
ECE Department
Polytechnic Institute of NYU
Brooklyn, NY, USA, 1120
msam01@students.poly.edu

Ozgur Sinanoglu
Faculty of Engineering
New York University - Abu Dhabi
Abu Dhabi, UAE, 129188
ozgursin@nyu.edu

Ramesh Karri
ECE Department
Polytechnic Institute of NYU
Brooklyn, NY, USA, 1120
rkarri@poly.edu

ABSTRACT

Camouflaging is a layout-level technique that hampers an attacker from reverse engineering by introducing, in one embodiment, dummy contacts into the layout. By using a mix of real and dummy contacts, one can camouflage a standard cell whose functionality can be one of many. If an attacker cannot resolve the functionality of a camouflaged gate, he/she will extract an incorrect netlist.

In this paper, we analyze the feasibility of identifying the functionality of camouflaged gates. We also propose techniques to make the dummy contact-based IC camouflaging technique resilient to reverse engineering. Furthermore, we judiciously select gates to camouflage by using techniques which ensure that the outputs of the extracted netlist are controllably corrupted. The techniques leverage IC testing principles such as justification and sensitization. The proposed techniques are evaluated using ISCAS benchmark circuits and OpenSparc T1 microprocessor controllers.

Categories and Subject Descriptors

B.2 [**Arithmetic and Logic Structures**]: Miscellaneous; B.6 [**Logic Design**]: Miscellaneous; B.7 [**Integrated Circuits**]: Miscellaneous

Keywords

IC reverse engineering; IC camouflaging; IP protection; IP piracy

1. INTRODUCTION

1.1 Reverse engineering of Integrated Circuits (ICs)

Reverse engineering of an IC is a process of identifying its structure, design and functionality. Traditionally, reverse engineering of IC's has been performed to collect competitive intelligence, to verify a design, and to check for commercial piracy and patent infringements. Reverse engineering can

CCS'13, November 4–8, 2013, Berlin, Germany.

http://dx.doi.org/10.1145/2508859.2516656.

- **identify the device technology** used in the IC. For instance, it was identified that Intel's Xeon processors use trigate transistors [1].
- **extract** the gate-level netlist[1] of the design. The gate-level netlist of a baseband processor from Texas Instruments was extracted in [2].
- **infer the functionality**. Reverse engineering on Apple's processor [3] revealed the type of graphic processing units used in iPhone 5.

Several techniques and tools have been developed to enable reverse engineering of an IC. [4] provides a tutorial on reverse engineering. Chipworks [5] and Degate [6] provide software products for reverse engineering. Shrinking device dimensions has not hampered reverse engineering. For instance, Intel's 22nm Xeon processor has been successfully reverse engineered [1].

While reverse engineering serves several benefits, an attacker can misuse it to steal and/or pirate a design. One can use the readily available tools and techniques for reverse engineering. On identifying a device technology, one can fabricate similar devices. One can extract a gate-level netlist and use it to design one's own IC or illegally sell it as an IP. Also, one can use the components extracted from competitor's products. This way one can reveal competitor's trade secrets. Because of these harmful effects, reverse engineering

[1]A gate-level netlist of a design consists of a set of interconnected logic gates such as NAND, NOR, AND, OR, XOR, XNOR, inverters, and memory elements such as flip-flops.

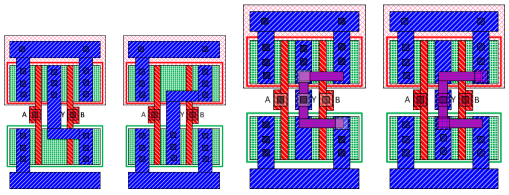


Figure 1: Standard cell layout of regular 2-input (a) NAND and (b) NOR gates. The metal layers are different and hence it is easy to differentiate them by just looking at the top metal layer. Camouflaged standard cell layouts of 2-input (c) NAND and (b) NOR gates. The metal layers are identical and hence it is difficult to differentiate them by just looking at the top metal layer.

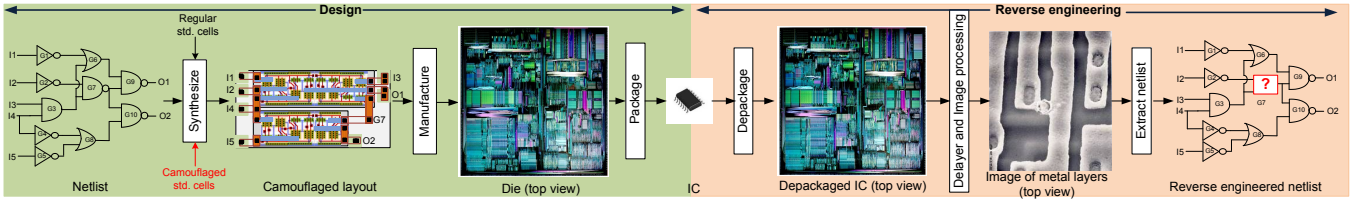


Figure 2: A design is synthesized into layout by using both regular standard cells and camouflaged standard cells. This layout is manufactured to obtain the IC. An attacker buys the camouflaged IC and depackages it. He/she delayers and images the metal layers and transistors. He/she then obtains the gate-level netlist by processing those images. However, the functionality of camouflaged cells cannot be resolved. For example, the functionality of G7 cannot be resolved.

was listed as one of the serious threats to semiconductor industry [7].

Detecting IP piracy by verifying competitor's IC requires that IP owners have access to that competitor's IC. Such a passive approach is bound to fail as potential adversaries can be anywhere across the globe. Proactive solutions that prevent, rather than detect, reverse engineering are preferable. For example, SypherMedia [8] provides IC camouflaging services for this purpose.

1.2 IC camouflaging to thwart reverse engineering

In one embodiment of IC camouflaging, the layouts of logic gates are designed to look identical, resulting in an incorrect extraction. For example, the layout of regular NAND cell (Figure 1(a)) and NOR ((Figure 1(b)) cell look different and are hence easy to reverse engineer. However, the layout of camouflaged NAND cell (Figure 1(c)) and NOR cell (Figure 1(d)) look identical and are difficult to differentiate [8–11]. When deceived into incorrectly interpreting the functionality of the camouflaged gate, the attacker may obtain a reverse engineered netlist that is different from the original. The netlist obtained by an attacker is the *deceiving netlist* where the functionality of the camouflaged gates are arbitrarily assigned.

Figure 2 shows how camouflaging protects an IC design against reverse engineering. A designer camouflages certain gates in the design[2]. For example, the NAND gate, G7, in Figure 2 is camouflaged. This design with camouflaged gates is then manufactured at a foundry. The manufactured IC is sold in the market. An attacker reverse engineers an IC by depackaging, delayering, imaging the layers, and extracting the netlist. However, in the extracted netlist, the functionality of the camouflaged gates are unknown. For example, in Figure 2, the functionality of G7 is unknown. An attacker assigns an arbitrary two-input function to it. Consequently, he may obtain an incorrect netlist.

To thwart reverse engineering of an IC, any camouflaging technique has to provide the following guarantees.
(1) **Resiliency to reverse engineering:** An attacker should not be able to identify the functionality of a camouflaged gate.
(2) **Corrupted outputs:** The outputs of the original and the deceiving netlists should be controllably different.

In this paper, we analyze the feasibility of identifying the functionality of the camouflaged gates. We propose an IC camouflaging technique that is resilient to reverse engineering. Furthermore, we judiciously select gates to camouflage by using techniques which ensure that the outputs of the deceiving netlist are controllably corrupted.

[2]A designer does not camouflage all the gates in the design because of the power, area, and delay overheads of the camouflaged gates. For example, the camouflaged NAND and NOR gates shown in Figure 1(c) and (d) are 1.5X bigger than their regular counterparts.

1.3 Contributions

The contributions[3] of this paper are:

- reverse engineering that can resolve the functionality of the camouflaged gates,
- a metric to measure the hardness of reverse engineering,
- a first technique to select the gates in a design to camouflage so that the functionality of the camouflaged gates can only be resolved by brute force,
- a second technique to select the gates to camouflage such that the deceiving netlist produces outputs which are controllably different from those of the original netlist, and
- evaluation of the proposed attack and defense techniques on OpenSparc T1 microprocessor [12].

The paper is organized as follows: Section 2 explains IC camouflaging and its security capabilities in detail. Section 3 describes the proposed techniques to reverse engineer a camouflaged IC. Section 4 details the proposed ways to select gates for camouflaging that is resilient to reverse engineering and can control the corruption at the outputs. Results of the proposed techniques on ISCAS benchmark circuits and controllers of OpenSparc processors are given in Section 5. Related work on IC reverse engineering and IP protection is summarized in Section 6. Section 7 concludes the paper.

2. IC CAMOUFLAGING

2.1 Steps in reverse engineering

IC reverse engineering involves the following steps [2].

- **Depackaging** the IC using corrosive chemicals.
- **Delayering** individual layers (metal, poly or diffusion) using corrosive chemicals.
- **Imaging** the top-view of each layer using an optical microscope or single electron microscope (SEM). The image may contain metal routing, contacts, vias and pins in that layer.
- **Annotation** aligns and stitches the images of different layers.
- **Extraction of gate-level netlist** from the annoted images. One can use the tools from Chipworks [5] and Degate [6] for this purpose.

[3]We are not proposing a new camouflaging technique. We are only improving the strength of existing techniques using principles from VLSI testing.

A reverse engineer may face the following difficulties.

Difficulty 1: Delayering the lower metal layers (M1 and M2) is difficult as compared to delayering higher metal layers (M3 and above) because lower metal layers are only a few tens of nanometers thick. Hence, a reverse engineer has to precisely control the strength of the chemicals used for delayering. Notwithstanding this difficulty, reverse engineers have successfully extracted information from the lower metal layers [2].

Difficulty 2: A reverse engineer can try to differentiate between a true and a dummy contact by slicing the die and imaging the side-view. However, there are hundreds of millions of contacts in an IC and an attacker has to slice the IC into million pieces to classify all of them. Hence, such reverse engineering will not be feasible.

Difficulty 3: A reverse engineer can use anisotropic techniques like reactive-ion etching to partially etch the layers. However, on using such techniques for top-down reverse engineering, by the time one reaches the bottom layers (where dummy contacts are placed), the dummy contacts are mostly eroded by the chemicals used to etch the upper layers. One may not know whether a broken/dummy contact is because of chemical erosion or camouflaging [13, 14].

2.2 Camouflaging using dummy contacts

The reverse engineer's inability to partially etch a layer is the basis for *dummy contacts*-based camouflaging [11]. Contacts are conducting materials that connect two adjacent metal layers or a metal layer 1 and poly. They pass through the dielectric that separates the two connecting layers. While the true contact has no gap, a dummy contact has a gap in the middle and fakes a connection between the layers.

Figure 3 shows the different layers of a logic gate with true and dummy contacts between M1 and poly. A true contact (Figure 3(a)) spans the entire dielectric and thus represents an electrical connection. However, a dummy contact (Figure 3(b)) has no electrical connection because of the gap. From the top view of the chip, both the true and dummy contacts appear identical under a microscope as shown in Figure 3.

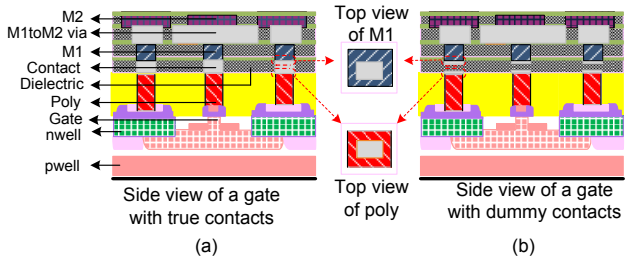


Figure 3: (a) A logic gate with true contacts and (b) a logic gate with dummy contacts. The top view, which is used by an attacker in reverse engineering, is identical for both the true and dummy contacts.

Table 1: List of true and dummy contacts to realize different functions using the camouflaged layout shown in Figure 4

Function	Contacts	
	True	Dummy
NAND	2, 4, 6, 8, 11, 12, 16, 17	1, 3, 5, 7, 9, 10, 13, 14, 15, 18, 19
NOR	2, 5, 6, 11, 12, 18, 19	1, 3, 4, 7, 8, 9, 10, 13, 14, 15, 16, 17
XOR	1, 3, 4, 7, 9, 10, 12, 13, 14, 15, 18, 19	2, 5, 6, 8, 11, 16, 17

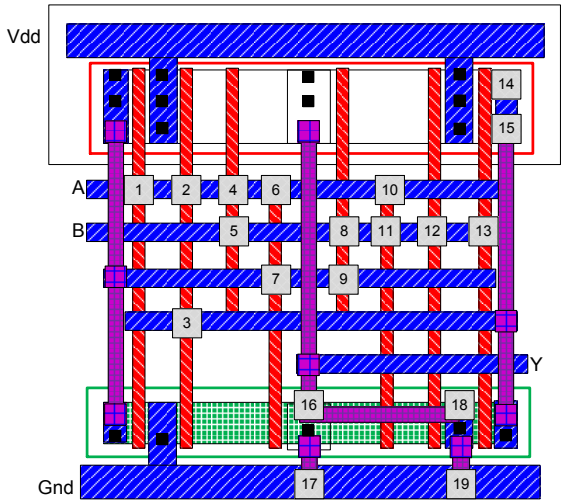


Figure 4: A generic camouflaged layout that can perform either as an XOR, NAND or NOR gate based on which contacts are true and dummy. The numbered boxes are the contacts.

2.3 Camouflaging standard cells

Standard cells perform a logic function and are used as building blocks in designing ICs. By using a mix of true and dummy contacts, one can design a standard cell whose functionality can be one of many. The true contacts dictate the functionality of a camouflaged gate. The use of optical and electrical microscopy will fail to identify the hidden functionality as they cannot differentiate between true and dummy contacts, complicating reverse engineering.

The more functionalities a camouflaged standard cell could implement, the more difficult the reverse engineering becomes. It is essential for a camouflaged standard cell to have a large number of transistors that can be connected in different ways to realize different logic functions. Of all the cells in the standard cell library, XOR and XNOR gates have the highest number of transistors (12). Hence, we choose to modify the layout of XOR and XNOR gates for camouflaging.

Figure 4 shows the layout of a camouflaged cell that can function as either 2-input XOR, NAND, or NOR. The sets of true and dummy contacts to implement different functions with the camouflaged gate are listed in Table 1. Similarly, one can design a camouflaged cell that can function as either 2-input XNOR, NAND or NOR.

3. REVERSE ENGINEERING A CAMOUFLAGED IC

3.1 Threat model

The attacker

1. has tools to reverse engineer an IC, i.e., the setup to delayer an IC, optical microscope or SEM to image the layers, and image processing software tools [5, 6].
2. can differentiate between a camouflaged standard cell and a regular standard cell from the images of different layers. This is possible because the images of regular and camouflaged standard cells are publicly available [8].
3. knows the list of functions that a camouflaged cell can implement. In our case, a camouflaged cell can implement one of {XOR, NAND, NOR} functions.

The objective of a reverse engineer is to determine the function implemented by every camouflaged gate. To reverse engineer a camouflaged IC, one performs the following steps.

1. Buys two copies of the target chip from the market.
2. Extracts the netlist with camouflaged gates from the first chip using the techniques listed in Section 2.1.
3. Computes input patterns using techniques from Section 3.2.
4. Applies these patterns on the second chip and obtains the outputs.
5. Resolves the functionality of camouflaged gates.

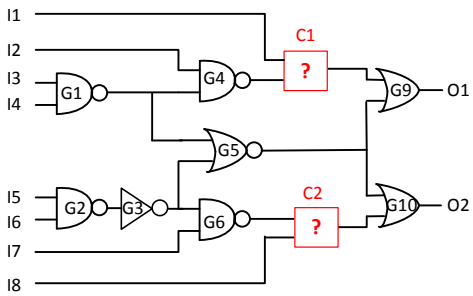


Figure 5: C1 and C2 are isolated camouflaged gates. An attacker can resolve the functionality C1 and C2 independent of each other.

3.2 Reverse engineering strategies

The critical step above is to generate the input patterns that help resolve the functionality of the camouflaged gates. The following two principles form the foundations in VLSI testing of manufacturing defects in ICs [15, 16].

Justification: The output of a gate can be justified to a known value by controlling one or more of its inputs. For example, an AND gate's output can be justified to '0' by setting one of its inputs to '0.'

Sensitization: A net can be sensitized to an output by setting all the side inputs of each gate in between to the non-controlling value of the gate. This way the value on the net is bijectively mapped to the value on the output.

3.2.1 Smart reverse engineering

Scenario: The attacker has completed the basic reverse engineering steps shown in Figure 2 and has the extracted netlist with regular and camouflaged gates. The critical step is to identify the functionality of the camouflaged gates. Each camouflaged gate could implement one of {XOR, NAND, NOR} functions. The attacker

1. justifies the inputs of the camouflaged gates to various two-input combinations, and
2. sensitizes their outputs to one of the primary outputs[4] to identify the response of the camouflaged gates to the two-input combinations.

[4]Primary outputs are output ports of the design which are accessible to an attacker.

Example 1: Consider the camouflaged gate C1 in Figure 5. The functionality of C1 can be resolved to be XOR by applying '010XXXX' at the inputs. This input pattern will justify the inputs of C1 to '00' and sensitizes the output of C1 to O1. If O1 is '0', then the functionality of C1 is resolved as XOR. Otherwise, the functionality of C1 can be resolved to be NOR by applying '110XXXX' at the inputs. This input pattern will justify the inputs of C1 to '10' and sensitize the output of C1 to O1. If O1 is '0', then the functionality of C1 is resolved as NOR. Otherwise, the functionality of C1 is resolved as NAND.

Camouflaged gates that do not have any circuit path interfering with other camouflaged gates are called *isolated* camouflaged gates. Example 1 illustrated how to attack isolated camouflaged gates. The attack gets complicated when the camouflaged gates interfere with each other. This is due to the complications arising from justification and sensitization in the presence of multiple camouflaged gates. We will illustrate this interference next.

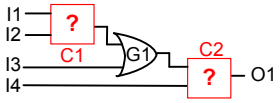


Figure 6: Interference between camouflaged gates, C1 and C2. An attacker targets C2 before targeting C1

3.2.2 Interference between camouflaged gates

Interferences between two camouflaged gates, C1 and C2, occur when C2 lies on a path between C1 and an output, and/or C1 and C2 converge at some other gate.

Smart reverse engineering: To identify the functionality of C2, an attacker has to find an input pattern that can justify the output of C1 to a known value and simultaneously resolve the functionality of C2 with the help of this known value.

Example 2: Consider the gates, C1 and C2, in Figure 6. The functionality of C1 can be one of {XOR, NAND, NOR}. Similarly, the functionality of C2 can be one of {XOR, NAND, NOR}. The functionality of C2 can be resolved to be an XOR by applying '1100' at the inputs. This pattern will justify the output of C1 to '0' irrespective of whether it functions as an XOR, NAND, or NOR, justify the inputs of C2 to '00,' and sensitize the output of C2 to O1. If O1 is 0, then the functionality of C2 is resolved as XOR.

Otherwise, the functionality of C2 can be resolved to be NOR by applying '1101' at the inputs. This pattern will justify the output of C1 to '0' irrespective of whether it functions as an XOR, NAND, or NOR, justify the inputs of C2 to '01,' and sensitize the output of C2 to O1. If O1 is 0, then the functionality of C2 is resolved as NOR. Otherwise, the functionality of C2 is resolved as NAND. After resolving the functionality of C2, C1 becomes an isolated camouflaged gate and its functionality can be resolved using the procedure specified for isolated camouflaged gates.

3.3 Limitations of smart reverse engineering

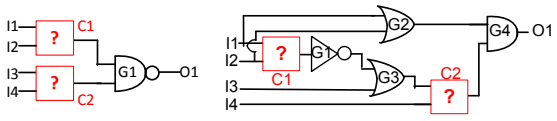


Figure 7: (a) C1 and C2 are non-resolvable camouflaged gates, (b) C2 is a partially resolvable camouflaged gate.

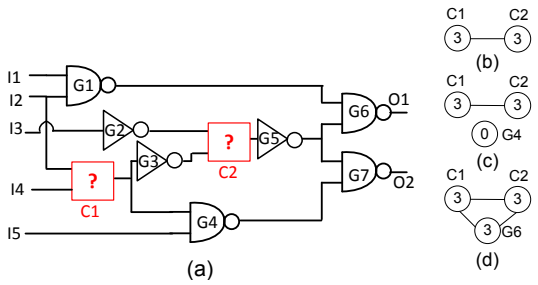


Figure 8: (a) A design camouflaged with 2 gates: C1 and C2, (b) Corresponding interference graph, (c) Interference graph with C1, C2, and G4 camouflaged, and (d) Interference graph with C1, C2, and G6 camouflaged. The number below the node label indicates the weight of a node.

While targeting a camouflaged gate, one cannot always generate an input pattern that can simultaneously justify the outputs of the other camouflaged gates to known values and resolve the functionality of the target gate. Consequently, an attacker cannot resolve the functionality of the camouflaged gates. Such gates are called non-resolvable camouflaged gates.

Example 3: Consider the camouflaged gates, C1 and C2, in Figure 7(a). If an attacker is trying to resolve C1, one needs to justify the output of C2 to '0', which is the only value that an XOR/NAND/NOR gate would produce for the input combination '11' non-ambiguously. However, setting C2's output to 0 will block C1's output from sensitizing to O1. Consequently, an attacker cannot resolve the functionality of C1. Similarly, an attacker cannot resolve the functionality of C2 in the presence of C1.

Sometimes an attacker can only partially resolve the functionality of a gate. Such gates are called partially resolvable camouflaged gates.

Example 4: Consider the camouflaged gates, C1 and C2, in Figure 7(b). If an attacker is trying to resolve that C2 is XOR, one needs to justify its inputs to '00' and justify C1's output to a known value ('0'). However, this is not possible as these two conditions are contradicting. On the contrary, if one is trying to resolve if C1 is NOR, one can use the input pattern 'X110'. This pattern will justify C2's inputs to '10', sensitize its output to O1, and justify C1's output to a known value ('0'). If O1's value is '1', C2 is resolved to be NOR. Otherwise, C2 functions as either NAND or XOR. However, an attacker cannot resolve the latter case. Thus, C2's functionality is only partially resolved.

3.4 Algorithm for attack

An attacker uses Algorithm 1 to resolve the functionality of the camouflaged gates. The attacker first identifies the isolated camouflaged gates and targets them. After that, one targets the resolvable interfering camouflaged gates. Only if one is able to generate an input pattern that simultaneously justifies the inputs of the target gate to the desired two-input combinations, justifies the output of other camouflaged gates to known values, and sensitizes the target gate's output to a primary output, the functionality of the target gate can be fully resolved. Finally, the functionality of non-resolvable and partially resolvable camouflaged gates are identified via brute force. As the functionality of a camouflaged gate is identified gradually in every iteration, it is replaced by the logic gate that performs the identified function. The overall computational effort depends on the amount of brute force applied.

Input : Camouflaged netlist, Functional IC
Output: Original netlist
Determine *Camouflaged gates*;
For each *remaining Camouflaged gate* **do**
  For each *Isolated camouflaged gate* **do**
    Compute and apply input pattern;
    Resolve *functionality* and update Netlist;
  end
  For each *Resolvable camouflaged gate* **do**
    if *there exists an input pattern* **then**
      Apply the input pattern;
      Resolve *functionality*, Update Netlist, Break;
    else
      ApplyBruteForce(), Break;
    end
  end
  For each *Partially resolvable camouflaged gate* **do**
    if *there exists an input pattern* **then**
      Apply the input pattern;
      Resolve partial *functionality*, Update Netlist, Break;
    else
      ApplyBruteForce(), Break;
    end
  end
  For each *Non-resolvable camouflaged gate* **do**
    ApplyBruteForce(), Break;
  end
end

ApplyBruteForce();
For each *possible combination of logic functions* **do**
  Generate random input patterns;
  Simulate the patterns and obtain the outputs OP_{sim};
  Apply the patterns on IC and obtain the outputs OP_{exe};
  if OP_{sim}==OP_{exe} **then**
    Functionality of camouflaged gates = current combination of logic functions;
    Update netlist;
  end
end

Algorithm 1: Reverse engineering a camouflaged IC

4. REVERSE ENGINEERING RESILIENT IC CAMOUFLAGING

Strong IC camouflaging hinges on camouflaging gates with complex interferences among them. Next, using a graph-based notation, we relate the types of camouflaged gates to the degree of interference they introduce.

4.1 Interference graph

We introduce an *interference graph* to drive the selection of gates for camouflaging. In this graph, each node represents a camouflaged gate. An edge connects two nodes if the corresponding camouflaged gates interfere with each other. Each node has a weight based on its type. Isolated and fully resolvable camouflaged gates are represented as isolated nodes with zero weights. Partially resolvable camouflaged gates have a weight of two as an ambiguity between two different functions needs to be resolved. Non-resolvable camouflaged gates have a weight of three as an ambiguity between three different functions needs to be resolved.

The maximum weight of the clique in the interference graph determines the strength of camouflaging, as the clique weight dictates the amount of brute force required to identify the functionality of camouflaged gates. Hence, a defender prefers to camouflage gates such that the interference graph is fully connected with each node having a weight of three.

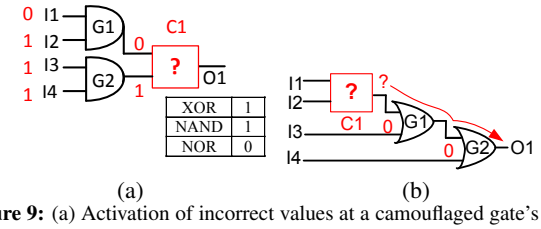


Figure 9: (a) Activation of incorrect values at a camouflaged gate's output and (b) propagation of incorrect values from C1's output to O1.

Example 5: Consider the circuit shown in Figure 8(a). In this circuit, C1 and C2 are non-resolvable camouflaged gates and interfere with each other. Hence, their nodes have a weight of 3 and are connected by an edge as shown in Figure 8(b). If G4 is selected as the third gate to be camouflaged, then it creates an isolated node in the interference graph. This is because an attacker can use the patterns 'X1010' and 'X1011' to resolve the functionality of G4. However, if G6 is selected as the third gate to be camouflaged, then it creates a non-resolvable node in the interference graph. Furthermore, it will be connected to the nodes of C1 and C2. This will increase the maximum clique size from 6 to 9. Thus, it is better to camouflage G6 than G4.

4.2 Are non-resolvable nodes enough for camouflaging?

IC camouflaging based on non-resolvable nodes ensure that an attacker cannot resolve the functionality of camouflaged gates. But, it does not ensure that the *deceiving netlist*[5] produces incorrect outputs. If the deceiving netlist produces correct outputs for most of the inputs, then camouflaging is effectively weak and the attacker will benefit. For minimum correlation, 50% output bits should differ for every input pattern[6].

Corrupting the outputs involves two principles: activation and propagation.

Activation is the process of justifying multiple input lines such that the output of a misinterpreted camouflaged gate is different from that of the gate in the original netlist, introducing an ambiguity.

Example 6: Consider the circuit shown in Figure 9(a). This circuit produces incorrect outputs for 15 out of the 16 possible input patterns. Only when C1 receives '11', it fails to introduce an ambiguity as NAND, NOR, and XOR produce the same value ('0'). In all the other cases, ambiguity is activated at C1, which is desired for output corruption.

Propagation is sensitizing a value to as many outputs as possible. In case of IC camouflaging, one has to propagate the ambiguity from the output of a camouflaged gate to as many outputs as possible.

Example 7: Consider the circuit shown in Figure 9(b). This circuit propagates the incorrect value of C1 to O1 for only 4 out of the 16 possible input patterns. In 3 of these 4 cases, ambiguity is activated.

4.3 Output corruptibility

We need maximum corruption, and thus minimum correlation at the outputs. The camouflaged gates should thus be selected judiciously. The output corruptibility function can be defined as:

$$\text{Output corruptibility} = \sum_{\#\text{ I/P patterns}} (\#\text{ O/Ps with ambiguity activated and propagated}) \quad (1)$$

Camouflaging the gate with the highest output corruptibility is expected to reduce the usability of the reverse engineered netlist.

4.4 Algorithm for smart camouflaging

Input : Original netlist, Number of gates to be camouflaged
Output: Camouflaged netlist
NumCamoGates = Number of gates to be camouflaged;
CamouflagedGates = {};
Calculate *OutputCorruptibility* for all gates;
SelectedGate = Gate with the highest *OutputCorruptibility*;
if *SelectedGate* $\notin$ *NAND||NOR||XOR* **then**
  Synthesize SelectedGate using NAND||NOR||XOR;
end
Camouflage SelectedGate;
CamouflagedGates += SelectedGate;
Construct InterferenceGraph;
for $i \leftarrow 2$ **to** *CamouflagedGates* **do**
  NonResolvableGates = {};
  For each $Gate_i$ *in Netlist* **do**
    if $Gate_i \notin$ *CamouflagedGates* **then**
      if $Gate_i$ *is Nonresolvable* **then**
        NonResolvableGates += $Gate_i$;
      end
    end
  end
  For each $Gate_j$ *in NonResolvableGates* **do**
    Compute *OutputCorruptibility*;
  end
  SelectedGate = NonResolvableGate with the highest *OutputCorruptibility*;
  if *SelectedGate* $\notin$ *NAND||NOR||XOR* **then**
    Synthesize SelectedGate using NAND||NOR||XOR;
  end
  Camouflage SelectedGate;
  CamouflagedGates += SelectedGate;
  Update InterferenceGraph;
end

Algorithm 2: Reverse engineering-resilient IC camouflaging

A defender can use the interference graph and the output corruptibility metric to camouflage gates as illustrated in Algorithm 2. Every iteration of the algorithm selects a gate that (1) is non-resolvable, (2) increases the maximum clique size in the interference graph, and (3) increases the output corruptibility metric. The interference graph is updated in every iteration. When a selected gate is not one of {XOR, NAND, NOR}, it is synthesized into one of these gates, so that it can be camouflaged. The algorithm terminates when a desired number of gates are camouflaged.

5. RESULTS

5.1 Experimental Setup

We considered two types of camouflaged gates. The first type can implement one of {XOR, NAND,NOR} function. The second type can implement an {XNOR, NAND,NOR} function. We compared the effectiveness of four types of ways to select a gate for camouflaging (1) *random*, where randomly selected gates are cam-

[5]Deceiving netlist is the netlist where camouflaged gates are incorrectly reverse engineered.

[6]0% means all the output bits are correct and implies direct correlation, while 100% implies inverse correlation.

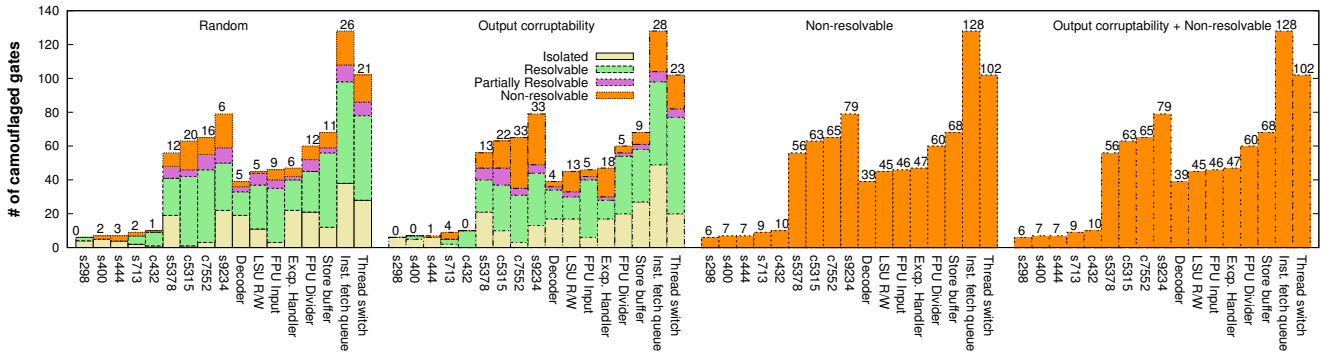


Figure 10: Types of camouflaged gates inserted by different ways of selecting gates to camouflage. Effective number of camouflaged gates for which an attacker will have to brute force are shown as numbers on top the bars. Note that *non-resolvable* and *output corruptibility + non-resolvable* selection techniques have same number of non-resolvable camouflaged gates (see discussion 1).

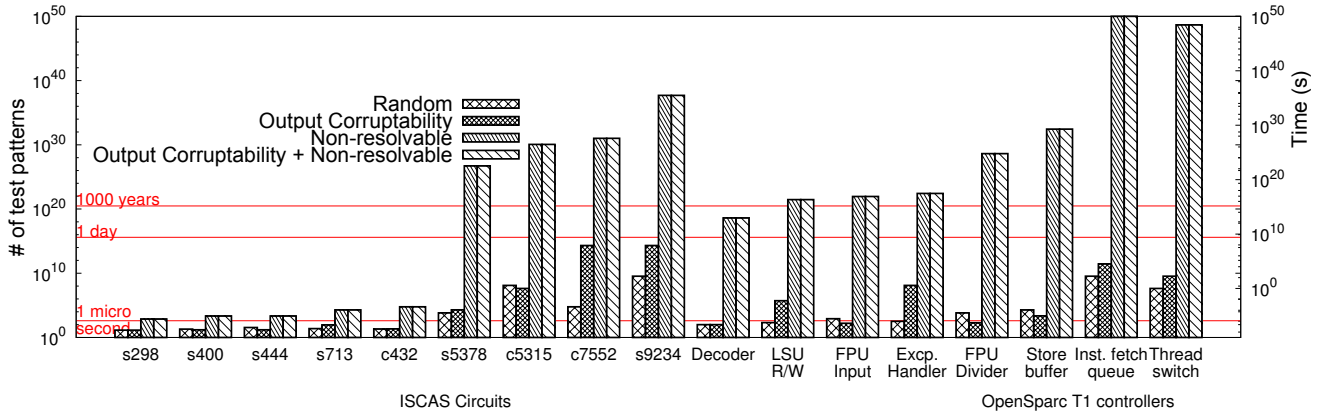


Figure 11: Number of possible test patterns that an attacker has to apply to resolve the functionality of camouflaged gates for different ways of selecting gates to camouflage.

ouflaged[7]; (2) *output corruptibility*, where the gates with high output corruptibility values are selected; (3) *non-resolvable*, where the gates are selected such that their functionality cannot be resolved; and (4) *output corruptibility + non-resolvable*, where the gates are selected using Algorithm 2.

These selection techniques are analyzed using ISCAS benchmark circuits [17] and OpenSparc T1 microprocessor controllers [12]. We used the HOPE fault simulation tool [18] to determine the input patterns for justification and sensitization for reverse engineering the functionality of a camouflaged gate. We used the HOPE tool to also calculate the output corruptibility by applying 1000 random input patterns. The Hamming distance between the outputs of the original netlist and the deceiving netlist was computed by applying 1000 random input patterns. To camouflage a circuit with a reasonable performance overhead, we camouflaged 5% of number of gates in that circuit. The area, power, and delay overheads are obtained using the Cadence RTL compiler [19].

5.2 Types of camouflaged gates and effective number of camouflaged gates

Figure 10 shows the number of each type of camouflaged gates for different ways in which gates to be camouflaged are selected. On *randomly* selecting gates, most of the gates are of isolated and resolvable types, benefiting the attacker. Only less than 20% of the camouflaged gates are either of partially resolvable or non-resolvable types, which require brute force. In *output corruptibility*-based selection, many of the gates are isolated to ensure corrupted outputs. An attacker can easily identify the functionality of camouflaged gates. In the *non-resolvable*- and *output corruptibility + non-resolvable*-based selections, all the camouflaged gates are non-resolvable, increasing the effort for an attacker.

Effective number of camouflaged gates: Since an attacker can always identify the functionality of isolated and fully resolvable gates, they do not cause any ambiguity to an attacker. In the case of partially resolvable camouflaged gates, the ambiguity is between two functions. In the case of non-resolvable camouflaged gates, the ambiguity is among three different functions. If there are 'N' partially resolvable and 'M' non-resolvable camouflaged gates, the total number of possible functions is $3^M \times 2^N$ for which an attacker has to brute force. The effective number of camouflaged gates is $log_3^{3^M \times 2^N} = M + Nlog_3^2$.

In Figure 10, the effective number of camouflaged gates are shown as numbers on top of the bars. The effective number of camouflaged gates of *random* and *output corruptibility* selections is less than that of *non-resolvable* and *output corruptibility + non-resolvable* selections. Therefore, the number of brute force attempts required to determine the functionality in *random* and *output corruptibility* selections is exponentially smaller than that of *non-resolvable* and *output corruptibility + non-resolvable* selections. In case of *random* and *output corruptibility* selections, the

[7]We chose random selection as the baseline as there is no publicly available algorithms to select gates to camouflage.

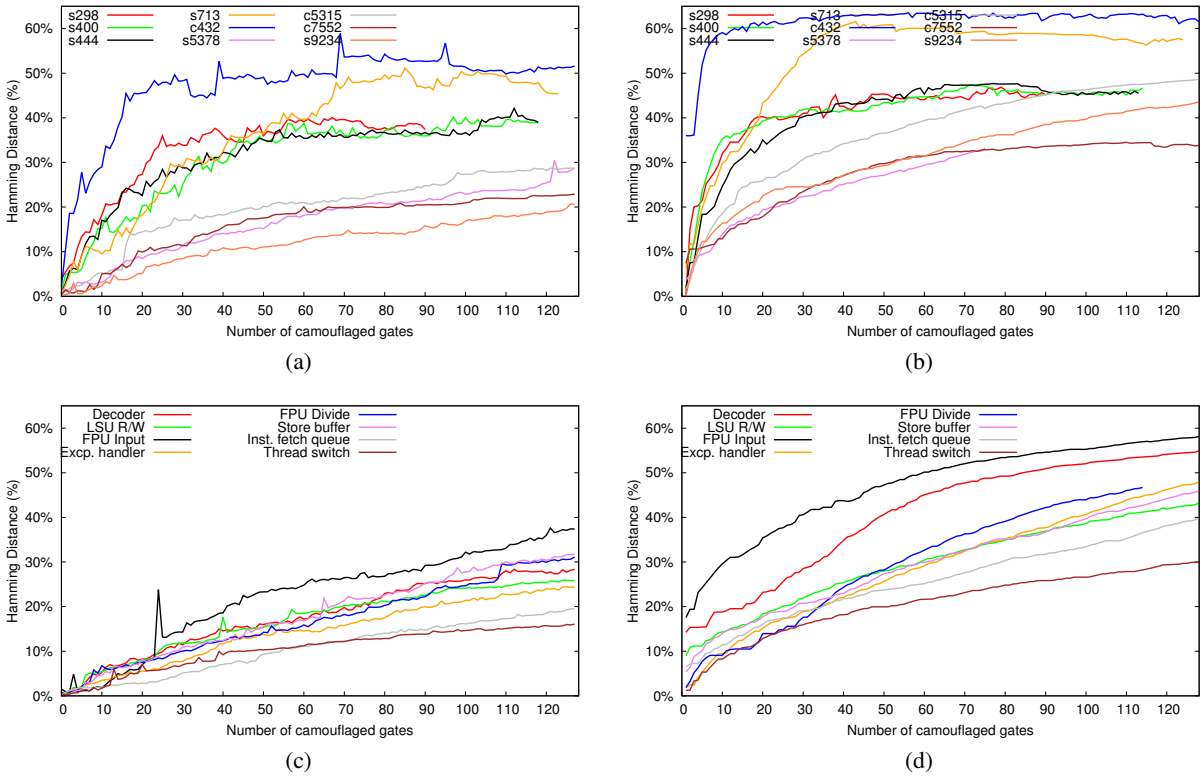


Figure 12: Hamming distance between the outputs of original and deceiving netlists. (a) Random camouflaging and (b) output corruptibility + non-resolvable selections on ISCAS circuits, (c) random camouflaging and (d) output corruptibility + non-resolvable selections on OpenSparc T1 controller modules.

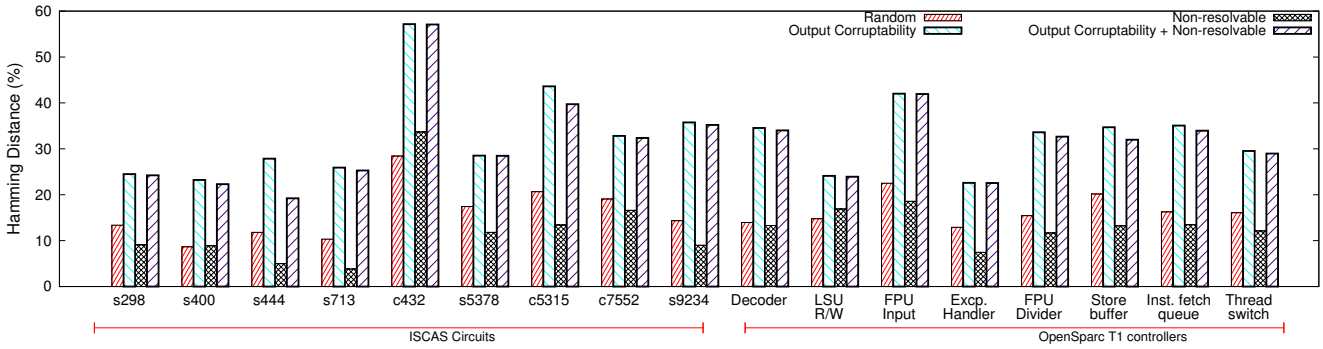


Figure 13: The best Hamming distance between the outputs of original and the deceiving netlists achieved when 5% of gates in the design are camouflaged for different ways of selecting gates to camouflage.

effective number of camouflaged gates is around one-fifth of the number of gates camouflaged. Thus for such selections, $4/5^{th}$ of the power, area, and delay budget spent on the camouflaged gates goes to waste.

5.3 Number of input patterns

Figure 11 shows the number of input patterns required to identify the functionality of camouflaged gates for different ways in which gates to be camouflaged are selected. The time scales are calculated assuming that an attacker can apply a billion patterns per second using a test equipment. *Non-resolvable*- and *output corruptibility* + *non-resolvable*-based selections require several thousands of years to break, even when only 40 gates are camouflaged.

5.4 Hamming distance analysis

Random selection is compared with *output corruptibility* + *non-resolvable*-based selection and the corresponding results are shown in Figure 12. When the gates are randomly selected, the Hamming distance between the outputs of original and deceiving netlists is less than that of *output corruptibility* + *non-resolvable*-based selection. Failure of activation and propagation is the main reason for this poor performance as discussed in Section 4.

The slope of the lines in Figure 12 indicate the performance of a selection technique. If the line is steeper, Hamming distance of desired level can be achieved with fewer camouflaged gates and hence, performance overhead will be smaller. As a result, *output corruptibility* + *non-resolvable*-based selection has smaller over-

head than randomly camouflaging gates as it camouflages fewer gates to achieve a target Hamming distance.

Figure 13 compares the Hamming distance achieved upon camouflaging 5% of the gates in the design for different selection techniques. It can been seen that *output corruptibility*- and *output corruptibility + non-resolvable*-based selections almost achieve similar Hamming distance. However, *output corruptibility*-based selection has a slightly better Hamming distance than *output corruptibility + non-resolvable*-based selection because the latter has another objective to satisfy in addition. However, both of these selections perform better than *random* and *non-resolvable*-based selection.

5.5 Power, area, and delay overheads

Table 2: Power, area and delay overheads for implementing a function using a camouflaged standard cell instead of a regular standard cell.

Function	Camouflaged gate					
	XOR+NAND+NOR			XNOR+NAND+NOR		
	Power	Delay	Area	Power	Delay	Area
NAND	5.5X	1.6X	4X	5.1X	1.8X	4X
NOR	5.1X	1.1X	4X	4.8X	1.4X	4X
XOR	0.8X	0	1.2X	N/A		
XNOR	N/A			0.7X	0	1.2X

Table 2 shows the power, area and delay overheads for implementing a function using a camouflaged standard cell instead of a regular standard cell. The power overhead is more than 5X because of the power consumed by the unused transistors. For example, a regular NAND gate uses only 4 transistors, but a camouflaged NAND gate uses 12 transistors. Furthermore, these transistors are bigger than the ones used in regular gates. Thus, the camouflaged cells consume more leakage and switching power, occupy a larger area, and are slower than the regular cells.

Figure 14 shows the power, area, and delay overheads on camouflaging 5% of the gates using *output corruptibility + non-resolvable*-selection technique. The power overhead is around 105% on an average. However, this power consumption is negligible when compared to that of the power consumption of the OpenSparc processor because the power consumption of a camouflaged controller is a few microwatts whereas the power consumption of a processor is few tens of watts [20, 21]. Camouflaging the read/write controller of load store unit (LSU R/W) has the highest area overhead of 135%. However, this unit has only 4000 gates when compared to 34 million gates in OpenSparc T1 [21,22]. Thus, camouflaging has a negligible overhead in the context of large designs such as processors. Nevertheless, the *output corruptibility + non-resolvable*-based selection ensures the budget spent on power, delay and area overhead is effective by preventing an attacker from reverse engineering; other selection techniques do not ensure such protection. To control this overhead, one can pursue a power and delay constrained camouflaging.

5.6 Discussion

1. Why is *output corruptibility + non-resolvable*-based selection better than *output corruptibility* and *non-resolvable*-based selections?
As shown in Figure 13, *output corruptibility*-based selection achieves a better Hamming distance than *output corruptibility + non-resolvable*-based selection. However, as shown in Figure 10, *output corruptibility*-based selection results in mostly isolated and fully resolvable camouflaged gates. Thus, an attacker can easily identify the functionality of the camouflaged gates. In fact, *output corruptibility*-based selection has more isolated and fully resolvable camouflaged gates than *random* selection. *Output corruptibility + non-resolvable*-based selection results only in non-resolvable camouflaged gates. Thus, an attacker cannot identify the functionality of the camouflaged gates.
As shown in Figure 10, both *non-resolvable* and *output corruptibility + non-resolvable*-based selections result in non-resolvable camouflaged gates. However, *output corruptibility + non-resolvable*-based selection has a better Hamming distance than *non-resolvable*-based selection as shown in Figure 13. In fact, *random* selection has a better Hamming distance than *non-resolvable*-based selection. To summarize, *output corruptibility + non-resolvable*-based selection is better than *output corruptibility* and *non-resolvable*-based selection techniques.

2. Can one use simpler camouflaged cells (for example, the ones shown in Figure 1(c) and (d)) instead of a complex camouflaged cell (for example, the one shown in Figure 4)?
A designer may use simpler and less costly camouflaged gates, such as those that supports just NAND and NOR functions. The overhead of this simple camouflaged gate will be much smaller than the XOR-based camouflaged gate. For instance, the camouflaged NAND and NOR cells shown in Figure 1(c) and (d) have a power overhead of 1.3X, area overhead of 1.5X, and delay overhead of 1.2X on average over their regular counterparts. The overhead cost is much smaller when compared to the overheads of XOR-based camouflaged gates listed in Table 2. However, from security perspective these simple gates have the following disadvantages over an XOR-based camouflaged gate.
(1) two instead of three possible functions reduce the amount of ambiguity for an attacker.
(2) two two-input combinations ('00' and '11') can non-ambiguously justify the output to a known value, making it easier for an attacker; an XOR-based camouflaged gate has only one two-input combination ('11') for non-ambiguous justification, and
(3) only two two-input combinations ('01' and '10') activate the ambiguity which results in reduced output corruptibility; an XOR-based camouflaged gate activates ambiguity for three two-input combinations ('00', '01,' and '10').
Thus, a simple camouflaged gate will have less resiliency to reverse engineering than XOR-based camouflaged gate.

3. Can one camouflage the entire design?
Camouflaging the entire design can make it difficult to resolve the functionality. However, it will tremendously increase the power, area, and delay overheads. Thus, a designer has to limit the number of camouflaged gates to honor the design's power, area, and delay budget. Our technique ensures that such budget-constrained camouflaging gates cannot be reverse engineered.

4. Why should one camouflage the controllers?
A design can be broadly classified into two parts – controller and datapath units. A designer's secret IP is typically realized in the form of controllers. Thus, they have to be protected from reverse engineering. Furthermore, controllers are usually tiny (<1%) [21, 23]. For example, OpenSPARC has 11.6×10^6 gates; The considered controllers have 1.2×10^4 gates. Thus, at the design level, the overhead due to camouflaging the controllers is negligible. Datapath units are often repetition of smaller blocks. Thus, camouflage costs would also be repeating.

5. Is Hamming distance the right metric?
Strength of cryptographic algorithms are often evaluated not only using the Hamming distance but also using metrics such as strict avalanche criterion (SAC) and bit independence criterion (BIC) [24]. While in this paper, we considered Hamming distance as the metric to measure output corruptibility, one can also use SAC

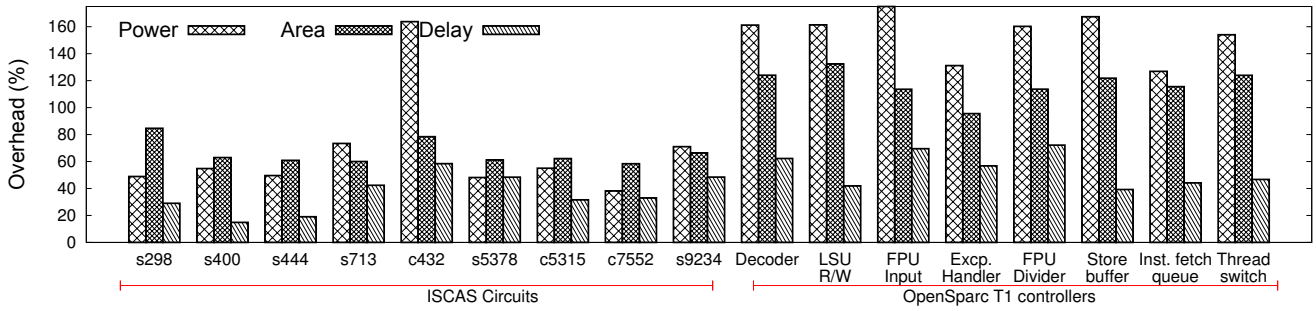


Figure 14: Power, area and delay overhead for camouflaging 5% of the gates using *output corruptibility + non-resolvable* selection.

and BIC for evaluation. However, calculating SAC and BIC becomes computationally infeasible for a design with a large number of inputs (>100). For example, the thread switch controller of OpenSparc T1 processor has 261 inputs. To calculate SAC and BIC, one has to apply 2^{261} input patterns. Hence, we choose the Hamming distance metric to analyze the effectiveness of our technique.

6. Can one use side-channel analysis to determine the functionality of camouflaged gates?
To prevent an attacker from resolving the functionality of a camouflaged gate using side-channel (power/delay) analysis, one can design a camouflaged gate that has similar power and delay characteristics independent of the functionality implemented by it.

6. RELATED WORK

6.1 Reverse engineering

IC reverse engineering techniques can be broadly classified into two types: extraction of gate-level netlist from layout, and extraction of functional specification from gate-level netlist. Reverse engineering of an IC to extract a gate-level netlist by performing the various steps shown in Figure 2 has been demostrated in [2,25]. An algorithm to extract a gate-level netlist from transistors has been presented in [26].

The DARPA IRIS program seeks to obtain the functional specification of a design by reverse engineering its gate-level netlist. Hansen et. al. exploited structural isomorphism to extract the functionality of datapath units [17]. Li et. al. reverse engineered the functionality of unknown units by performing behavioral matching against a library of components with known functionalities such as adders, counters, register files, and subtracters [27]. Subramanyan et. al. identified the functionality of unknown modules by performing Boolean satisfiability analysis with a library of components with known functionalities [28].

6.2 IC camouflaging

A design can be camouflaged by using lookalike standard cells [8]. These lookalike standard cells perform different logic functions while they have similar metal connections to deceive an attacker. Subtle changes are made in their layout to realize different logic functions. An attacker can look for these subtle changes in the layout to identify the functionality. The camouflaged gates considered in this paper look identical in all the metal layers but differ only in terms of true and dummy contacts, which cannot be distinguished by an attacker as discussed in Section 2.

An IC camouflaging technique can also leverage unused spaces in a chip and fill them with standard cells [13]. The outputs of these filler cells will not drive any active logic. One can also camouflage a design by using programmable standard cells [10]. Post-fabrication, these cells will be programmed using a control input. Similar to introducing dummy contacts, a designer can also create dummy channels, which will result in non-functional transistors [14]. Such non-functional transistors can be used to deceive an attacker. Using the techniques outlined in this paper, the security of all these camouflaging methods can be analyzed.

6.3 Hardware IP protection techniques

Logic obfuscation hides the functionality and implementation of a design by inserting additional components into it. Logic obfuscation techniques can be broadly classified into two types—sequential and combinational. In sequential logic obfuscation, additional logic (black) states are introduced in the finite state machine (FSM) of a design [23, 29]. The FSM is modified in such a way that the design reaches a valid state only on applying the correct key. If the key is withdrawn, the design ends up in a black state. In combinational logic obfuscation, XOR/XNOR gates are introduced to conceal the functionality of a design [30]. The circuit will function correctly only on applying the correct value to these gates. An algorithm to insert XOR/XNOR gates for a stronger obfuscation has been proposed in [31]. An algorithm to ensure incorrect outputs on applying a wrong key has been proposed in [32]. Obfuscation is also performed by inserting memory elements [33]. The circuit will function correctly only when these elements are programmed correctly. These techniques require tamper-proof and non-volatile memories to store keys on-chip and safe exchange of keys between the IP designer and the user. However, IC camouflaging does not encounter such issues as it does not use keys. IC camouflaging is a complementary technique to logic obfuscation; IC camouflaging operates at layout-level while logic obfuscation techniques operate at gate- and register transfer levels.

Techniques such as watermarking and passive metering are also proposed to detect IC piracy. In watermarking techniques, a designer embodies his/her signature into the design [34]. During litigation, the designer can reveal the watermark and claim ownership of an IC/IP. Watermarks are constructed by adding additional states to the finite state machine of the design, adding secret constraints during high-level [35], logical, or physical synthesis [36]. In passive metering techniques, a unique device ID for every IC is created by leveraging process variations [23]. Physical unclonable functions are leveraged to produce such IDs [37, 38]. If a user pirates an IC, he/she will be caught by tracking the device ID.

7. CONCLUSION

Although dummy contact-based IC camouflaging is an effective layout-level technique against reverse engineering, it is weak when the camouflaged gates are isolated and are fully resolvable. While

a designer can camouflage all the gates, it incurs power, delay and area overheads. We show that camouflaging can be strengthened by judiciously selecting the gates in the design to camouflage, without incurring much overhead. The proposed techniques rely on resolvability and corruptibility metrics that deliver camouflaging solutions which are resilient to reverse engineering. This way the attacker is forced to do brute force.While we formulated these security requirements as constraints for automatic test pattern generation (ATPG) tools, one cause also use SAT based ATPG and equivalence-checking tools exist to guide selection of gates for IC camouflaging. Reverse engineering becomes complicated when other camouflaging techniques such as programmable cells [10] and dummy filler cells [14] are used in conjunction with dummy contacts. The proposed security metric can be used to evaluate the strength of these camouflaging techniques either individually or when combined.

8. ACKNOWLEDGMENTS

The authors thank Dr. Ron Cocchi from SypherMedia International for his valuable comments.

9. REFERENCES

[1] Chipworks, "Intel's 22-nm Tri-gate Transistors Exposed," http://www.chipworks.com/blog/technologyblog/2012/04/23/intels-22-nm-tri-gate-transistors-exposed/, 2012.

[2] R. Torrance and D. James, "The state-of-the-art in semiconductor reverse engineering," *in the Proc. of IEEE/ACM Design Automation Conference*, pp. 333–338, 2011.

[3] ExtremeTech, "iPhone 5 A6 SoC reverse engineered, reveals rare hand-made custom CPU, and tri-core GPU," http://www.extremetech.com/computing/136749-iphone-5-a6-soc-reverse-engineered-reveals-rare-hand-made-custom-cpu-and-a-tri-core-gpu.

[4] Silicon Zoo, "The layman's guide to ic reverse engineering," http://siliconzoo.org/tutorial.html.

[5] Chipworks, "Reverse engineering software," http://www.chipworks.com/en/technical-competitive-analysis/resources/reerse-engineering-software.

[6] Degate, http://www.degate.org/documentation/.

[7] SEMI, "Innovation is at risk as semiconductor equipment and materials industry loses up to $4 billion annually due to IP infringement," www.semi.org/en/Press/P043775, 2008.

[8] SypherMedia, "Syphermedia library circuit camouflage technology," http://www.smi.tv/solutions.htm.

[9] J. P. Baukus, L. W. Chow, R. P. Cocchi, and B. J. Wang, "Method and apparatus for camouflaging a standard cell based integrated circuit with micro circuits and post processing," *US Patent no. 20120139582*, 2012.

[10] J. P. Baukus, L. W. Chow, R. P. Cocchi, P. Ouyang, and B. J. Wang, "Building block for a secure cmos logic cell library," *US Patent no. 8111089*, 2012.

[11] J. P. Baukus, L. W. Chow, and W. Clark, "Integrated circuits protected against reverse engineering and method for fabricating the same using an apparent metal contact line terminating on field oxide," *US Patent no. 20020096776*, 2002.

[12] "Sun Microsystems, OpenSPARC T1 Processor," *http://www.opensparc.net/opensparc-t1/index.html*.

[13] J. P. Baukus, L. W. Chow, R. P. Cocchi, P. Ouyang, and B. J. Wang, "Camouflaging a standard cell based integrated circuit," *US Patent no. 8151235*, 2012.

[14] J. P. Baukus, L.-W. Chow, J. W. M. Clark, and G. J. Harbison, "Conductive channel pseudo block process and circuit to inhibit reverse engineering," *US Patent no. 8258583*, 2012.

[15] M. L. Bushnell and V. D. Agrawal, "Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits," *Kluwer Academic Publishers, Boston*, 2000.

[16] M. Abramovici, M. A. Breuer, and A. D. Friedman, "Digital Systems Testing & Testable Design," *Wiley*, 1994.

[17] M. Hansen, H. Yalcin, and J. Hayes, "Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering," *IEEE Design Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.

[18] H. Lee and D. S. Ha, "HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 9, pp. 1048–1058, 1996.

[19] Cadence, "RTL Compiler," www.cadence.com/products/ld/rtl_compiler.

[20] K. Constantinides, "Online low-cost defect tolerance solutions for microprocessor designs," web.eecs.umich.edu/ taustin/papers/Kypros_Thesis.pdf.

[21] A. Waksman, J. Eum, and S. Sethumadhavan, "Practical, lightweight secure inclusion of third-party intellectual property," *IEEE Design & Test*, no. 99, pp. 1–1, 2013.

[22] Oracle, "Opensparc internals," http://www.oracle.com/technetwork/systems/opensparc/opensparc-internals-book-1500271.pdf.

[23] Y. Alkabani and F. Koushanfar, "Active hardware metering for intellectual property protection and security," *in the Proc. of USENIX security*, pp. 291–306, 2007.

[24] H. Heys and S. Tavares, "Avalanche characteristics of substitution-permutation encryption networks," *IEEE Transactions on Computers*, vol. 44, no. 9, pp. 1131 –1139, 1995.

[25] R. Torrance and D. James, "The state-of-the-art in ic reverse engineering," *in the Proc. of Cryptographic Hardware and Embedded Systems*, pp. 363–381, 2009.

[26] W. M. V. Fleet and M. R. Dransfield, "Method of recovering a gate-level netlist from a transistor-level," *US Patent no. 6190433*, 1998.

[27] W. Li, Z. Wasson, and S. Seshia, "Reverse engineering circuits using behavioral pattern mining," *in the Proc. of IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 83–88, 2012.

[28] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, "Reverse engineering digital circuits using functional analysis," *in the Proc. of IEEE/ACM Design Automation and Test in Europe*, 2013.

[29] R. Chakraborty and S. Bhunia, "HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1493–1502, 2009.

[30] J. Roy, F. Koushanfar, and I. Markov, "EPIC: Ending Piracy of Integrated Circuits," *IEEE Computer*, vol. 43, no. 10, pp. 30–38, 2010.

[31] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Security analysis of logic obfuscation," *in the Proc. of IEEE/ACM Design Automation Conference*, pp. 83–89, 2012.

[32] ——, "Logic encryption: A fault analysis perspective," *IEEE Design, Automation Test in Europe*, pp. 953–958, 2012.

[33] A. Baumgarten, A. Tyagi, and J. Zambreno, "Preventing IC Piracy Using Reconfigurable Logic Barriers," *IEEE Design and Test of Computers*, vol. 27, no. 1, pp. 66–75, 2010.

[34] A. Kahng, J. Lach, W. Mangione-Smith, S. Mantik, I. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe, "Watermarking techniques for intellectual property protection," *in the Proc. of IEEE/ACM Design Automation Conference*, pp. 776–781, 1998.

[35] F. Koushanfar, I. Hong, and M. Potkonjak, "Behavioral synthesis techniques for intellectual property protection," *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, no. 3, pp. 523–545, 2005.

[36] A. Kahng, S. Mantik, I. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe, "Robust IP watermarking methodologies for physical design," *in the Proc. of IEEE/ACM Design Automation Conference*, pp. 782–787, 1998.

[37] G. Suh and S. Devadas, "Physical Unclonable Functions for Device Authentication and Secret Key Generation," *in the Proc. of the IEEE/ACM Design Automation Conference*, pp. 9–14, 2007.

[38] J. Lee, D. Lim, B. Gassend, G. Suh, M. van Dijk, and S. Devadas, "A technique to build a secret key in integrated circuits for identification and authentication applications," *in the Proc. of IEEE Internationall Symposium on VLSI Circuits*, pp. 176–179, 2004.

[39] Cadence, "SoC Encounter," http://www.cadence.com/products/di/soc_encounter/ pages/default.aspx.

APPENDIX

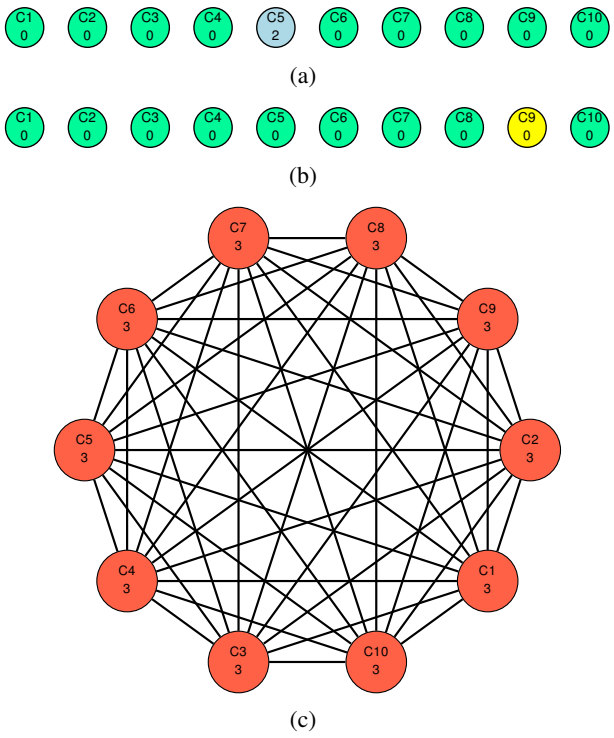
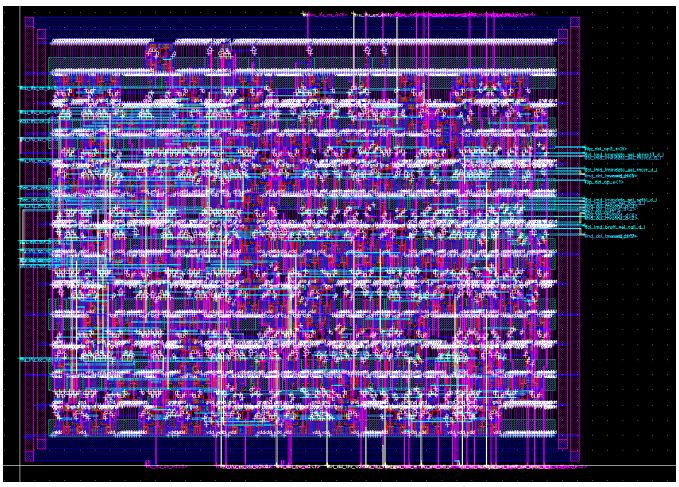


Figure 15: Interference graph for C432, one of the ISCAS benchmark circuits. This design has 36 inputs, 7 outputs, and 160 gates. 10 gates in this design are selected for camouflaging. The figure shows the interference graph when gates to camouflage are selected in three different ways. Nodes in green, yellow, blue, and red indicate isolated, fully resolvable, partially resolvable, and non-resolvable gates, respectively. The number below the node label indicates the weight of the node. (a) On *randomly* selecting gates, 9 gates are isolated and one gate is partially resolvable. In this interference graph, the number of brute force attempts required to resolve the functionality of camouflaged gates is 2. (b) When gates are selected using the*output corruptibility* metric, 9 gates are isolated and one gate is fully-resolvable. In this interference graph, the number of brute force attempts required to resolve the functionality of camouflaged gates is 0. (c) When gates are selected using the *output corruptibility + non-resolvability* metric, all the gates are non-resolvable. In this interference graph, the number of brute force attempts required to resolve the functionality of camouflaged gates is 3^{10}. When the gates are selected using the *non-resolvability* metric, the resultant interference graph is same the one in (c).

Controller	**Possible IP secret**	**# of gates**
Instruction queue	Cache line validation	2027
Instruction decoder	Decoding algorithm	771
Load Store unit read/write	Read/write priorities	899
Load store unit store buffer	Ordering of store operations	1360
Floating point input	Scheduling of FPU operations	916
Floating point divider	Division algorithm	1197
Exception handler	Supported exceptions	936
Trap logic unit	Interrupt handling & Machine states	2724

(a)



(b)

Figure 16: (a) Controllers of OpenSparc T1 microprocessor and their corresponding IP secret. (b) The layout with camouflaged gates for the controller unit of instruction decoder in OpenSparc T1 microprocessor. This design has 132 inputs, 89 outputs, 771 gates. 39 gates in this unit have been camouflaged. The design was synthesized for the library with camouflaged gates using Cadence RTL compiler [19]. The layout from the synthesized design was generated using Cadence SoC Encounter [39].