University of Massachusetts Amherst

ScholarWorks@UMass Amherst

Masters Theses                                                    Dissertations and Theses

October 2017

# ORACLE GUIDED INCREMENTAL SAT SOLVING TO REVERSE ENGINEER CAMOUFLAGED CIRCUITS

Xiangyu Zhang
*University of Massachusetts Amherst*

Follow this and additional works at: https://scholarworks.umass.edu/masters_theses_2

Part of the Digital Circuits Commons, and the VLSI and Circuits, Embedded and Hardware Systems Commons

# ORACLE GUIDED INCREMENTAL SAT SOLVING TO REVERSE ENGINEER CAMOUFLAGED CIRCUITS

A Thesis Presented

by

XIANGYU ZHANG

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2017

Electrical and Computer Engineering

# ORACLE GUIDED INCREMENTAL SAT SOLVING TO REVERSE ENGINEER CAMOUFLAGED CIRCUITS

A Thesis Presented

by

XIANGYU ZHANG

Approved as to style and content by:

_____

Daniel Holcomb, Chair

_____

Maciej Ciesielski, Member

_____

Sandip Kundu, Member

_____

Christopher V. Hollot, Head
Electrical and Computer Engineering

# DEDICATION

*In the name of Jesus Christ.*

# ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Daniel Holcomb, for his thoughtful, patient guidance and support. Thanks are also due to Duo Liu and Cunxi Yu. Together their friendship and selfless contribution to my professional development have been invaluable and will forever be appreciated. I would also like to extend my gratitude to the members of my committee, Dr. Sandip Kundu and Dr. Maciej J. Ciesielski, for their helpful comments and suggestions on all stages of this project.

A special thank you to all those whose support and friendship helped me to stay focused on this project and who have provided me with the encouragement to continue when the going got tough.

# ABSTRACT

## ORACLE GUIDED INCREMENTAL SAT SOLVING TO REVERSE ENGINEER CAMOUFLAGED CIRCUITS

SEPTEMBER 2017

XIANGYU ZHANG

B.Sc., FLORIDA INSTITUTE OF TECHNOLOGY

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Daniel Holcomb

This study comprises two tasks. The first is to implement gate-level circuit camouflage techniques. The second is to implement the Oracle-guided incremental de-camouflage algorithm and apply it to the camouflaged designs.

The circuit camouflage algorithms are implemented in Python, and the Oracle-guided incremental de-camouflage algorithm is implemented in C++. During this study, I evaluate the Oracle-guided de-camouflage tool (Solver, in short) performance by de-obfuscating the ISCAS-85 combinational benchmarks, which are camouflaged by the camouflage algorithms. The results show that Solver is able to efficiently de-obfuscate the ISCAS-85 benchmarks regardless of camouflaging style, and is able to do so 10.5x faster than the best existing approaches. And, based on Solver, this study also measures the de-obfuscation runtime for each camouflage style.

# CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

**Figure**      **Page**

# CHAPTER 1

# INTRODUCTION

IC designers have clear incentives against publicizing all implementation details of a design, as this may compromise their strategic advantage or leak sensitive information. However, once a circuit is fabricated and released to market, reverse engineering techniques can attempt to extract implementation details from the physical object without consent or knowledge of the designer. Circuit camouflaging is an attempt to obscure the true functionality of a circuit, and to limit the information that can be leaked through reverse engineering.

Gate-level camouflaging is a particular camouflaging technique in which the functions of certain combinational logic gates cannot be directly ascertained from imaging-based reverse engineering. In this case, the logic may be inferred using a combination of information obtained from reverse engineering and information obtained through observation of input-output vectors captured through scan chains or other mechanisms. In my work I present such an algorithm for extracting the functionality of reverse engineered netlists.

The specific contributions of this work are as follows:

- I present an incremental-SAT-based technique for reverse engineering camou-flaged integrated circuits that outperforms its non-incremental counterpart by 10.5x in terms of average runtime on ISCAS-85 benchmarks.

- I show that selective gate camouflaging based on the objective of maximizing output corruption [27] offers no resistance to reverse engineering and can reduce the number of vectors required to deobfuscate a circuit.

- I provide a new standard and widely-applicable tool for logic de-obfuscation that can be used to evaluate current and future approaches for selective camouflaging.[1]

- I demonstrate that our technique is general and can efficiently resolve the obfuscated function of three proposed camouflaging techniques [27, 23, 4].

---

[1]The source code of our tool and benchmarks used in this work are released publicly at our project website at
*https://ycunxi.github.io/Incremental-SAT-DeCam/.*

# CHAPTER 2

# RELATED WORK

By decapsulating chips, and removing layers and imaging in succession to reveal the internal information, invasive techniques can be used by a reverse engineer to steal gate-level circuit functions. Recently, such reverse engineering of integrated circuit (IC) chips has caused loss for the IC industry [32]. Torrance and James [32] gave an overview of the state of the art in invasive reverse engineering.

Recent years, people focus on extracting high-level meaning from the deluge of gates by invasive reverse engineering of fabricated circuits. Through matching against known components, Li managed to resolve subcircuit components and obtain word-level structures [20]. Subramanyan also improves to operate on the unstructured netlist, where subcircuits are not identified in advance [30]. Work by Gascón [10] managed to check equivalence between a circuit-under-investigation and a reference circuit when the signal correspondence between the two is unknown. Different from the above mentioned methods, there is no complete gate-level model of the circuit-under-investigation.

Camouflaged gates is a countermeasure against image-based reverse engineering. Camouflaged gates can provide the same looking cells while the logic functions or connections are substantially different. In such way, a circuit's function cannot be inferred from its appearance. Camouflaged gate libraries [31] can be realized by using hard-to-observe structural techniques to create different gate functions [6], or functionality can be controlled without structural differences by changing doping of specific devices [1, 28, 23, 12]. To minimize cost, one may choose to only camouflage

a small portion of the circuits [3, 27]. However, without the knowledge of the actual circuit, a reverse engineer needs to consider all gates are camouflaged [2]. Vijayakumar provides an overview of physical mechanisms for obfuscation [33].

Rajendran al. gives an attacker model for reverse engineering circuits with camouflaged gates [26]. When the attacker knows all non-camouflaged gates and can apply input and get output, the camouflaged circuit remain hard to discover. My work uses oracle-guided synthesis [13], but the capability and limitation of this technique need attention. Though the overall circuit produced by oracle-guided synthesis is guaranteed to be functionally equivalent to the obfuscated circuit, it is not guaranteed to have equivalence gate by gate. For example, given that 2-input XOR and 2-input NAND gates produce different outputs only for the 00 input combination, the two gate functions are interchangeable if the 00 input combination cannot be justified or propagated to the outputs. This gate-level ambiguity is unavoidable if trying to synthesize a design based on merely inputs and outputs, but it is important to note that a design recovered through oracle-guided synthesis should not be used for certain classes of side-channel attacks or fault injection attacks that require knowing the states of all combinational circuit nets.

During my work, I modeled all the three mainstream camouflage techniques by a logic model based on multiplexer, but the problem of logic locking arise. Logic locking is closely related to the problem of camouflaging and in logic locking, recent works show the importance of trying to thwart SAT attacks by ensuring that each input-output example provides limited information about the values of the key bits that unlock the circuit  [36] [34]. In an extreme case, one can guarantee that an exponential number of input-output examples are needed to exactly learn the key bits, but a consequence of this is that output corruption under incorrect key guesses will be limited [36]. A similar approach has been used with camouflaging to quantify the security of the camouflaged circuits [37] [19]. Note that in logic locking, care

must be taken to ensure that the key gates cannot be identified and removed, and thus to remain secure under reverse engineering logic locking can be combined with obfuscation [36] [15].

Reverse engineering could be viewed as a form of privacy threat with respect to the IP implemented on a chip. In a related reverse engineering context, privacy of specific chip instances can be violated by an ability to observe unintended unique features of each device instance [14, 25].

# CHAPTER 3

# CIRCUIT CAMOUFLAGE AND ATTACKER MODEL

The attacker model in my work is that of an attacker have instances of a chip and want to reconstruct the logic function of the whole chip. Based on prior works and the attacker model proposed by Rajendran [27], the assumption of the capability of the attacker is as follows:

1. By imaging the layers, an attacker can obtain the information of the logic functions and the topology of the circuit. Namely, the attacker can generate a netlist from such instances of a chip.

2. The attacker can identify which components are camouflaged, which is because the camouflaged parts do not look identical to the non-camouflaged parts. For example, a camouflaged gate that can be configured to a number of different possible functions is usually larger than the function-specific implementation of the same gates.

3. The attacker can apply input to the camouflaged combinational circuit and observe the output signals. Because attacker has access to an instance of chips, this could be achieved by using scan-based test techniques [35], possibly combined with micro probing to gain access to deactivated scan chains [17].

In this work, I model all the configurations of the camouflaged circuit using multiplexers with uncertain control bits, because an uncertainty of connection and functionality can be translated to uncertainty about the value of certain Boolean variables . In Chapter 4, I show how to solve for the value of these variables using

SAT. The resolved value can thus indicate the actual logic function or connection of the camouflaged gate. In this chapter, I will explain the principle of the three different types of camouflaged components: *Camouflaged Standard Cells*, *Obfusgates*, and *Transformable Interconnects*, as well as how to apply this modeling mechanism.



(a) NAND/NOR/XOR Camouflage          (b) Fully Camouflaged
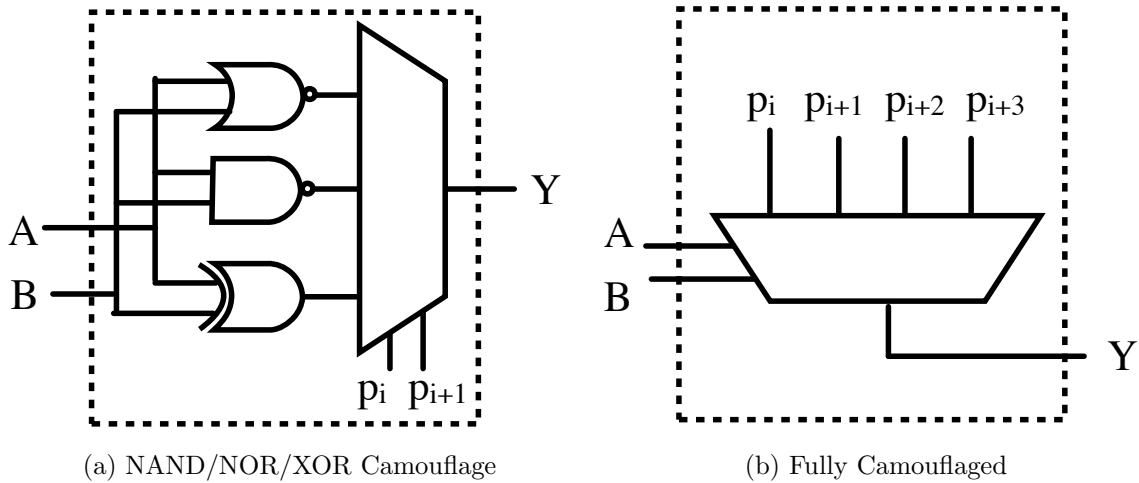
Figure 3.1: Circuit constructs used to model Camouflaged Standard Cells. Boolean variables $p_i$ configure the logic function of the gate, and an attacker tries to learn the value of these variables.

## 3.1  Camouflaged Standard Cells using Dummy Contacts

The principle of standard cell camouflaging is using a generic cell layout to realize multiple logic functions. Through some invisible changes to this layout, the functionality can be substantially changed while attacker cannot notice those changes from an imaging-based attack. One way to implement hard-to-observe changes is to use *Dummy Contacts* [5]. Dummy contacts are structures that look like connections between metal layers, or connections between metal layers and polysilicon, but there is no such connection. A same-looking cell can be configured to different logic functions depending on which contacts in a cell are true contacts. According to [5], *Dummy Contacts* can not be found by imaging-based technique, so a reverse engineer cannot know which logic function is implemented by the cell.

A specific variant of a camouflaged standard cell is a generic cell layout that can realize the functionality of 2-input XOR, NAND, or NOR gates depending on which contacts are true [27]. To model the camouflaged cell, I introduce a multiplexer-based model (Figure 3.1). The functionality can be derived by resolving the values of program bits $p_i, p_{i+1}$ (Figure 3.1 a). Note that in this model $p_i, p_{i+1}$ are forbidden to be "11" since there are only three possible functions for the gate. Additionally, I extend the model (Figure 3.1 b) to gain the capability of modeling other types of camouflaged standard cells. The function of $Y$ can be represented as $Y = \bar{A}\bar{B}p_1+\bar{A}Bp_2+A\bar{B}p_3+ABp_4$. Hence, this model is able to represent any of the sixteen possible two-input functions.

## 3.2   Obfusgates: Dopant Programmable Logic Cells

Similar to the principle of camouflaged standard cell, Malik et al. [23] proposed another indistinguishable component, which is *Obfusgates*. An Obfusgate is implemented by combining a standard cell logic gate with some *Obfuscells* connected to its input and output ports. Malik demonstrates Obfusgates based on both NAND4 and AND2 [23], but here I consider only the NAND4 variant. Depending on the dopant polarity within the active area of the Obfuscell, each Obfuscell can have four logic functions: inverter, buffer, constant 1, or constant 0. Because imaging-based technique can not see the dopant polarity, attackers will have difficulty learning the logical function of the Obfusgate [23].

Unlike camouflaged standard cells that can only provide functionality obfuscation, Obfusgates can provide both functionality and connection obfuscation. The functionality obfuscation is based on the different configuration of Obfuscell. And the connection obfuscation relies on the fact that pins connected to the constant 0 and constant 1 inputs can provide additional dummy wire, which can be connected to any sources without impacting the logic function of the circuit.
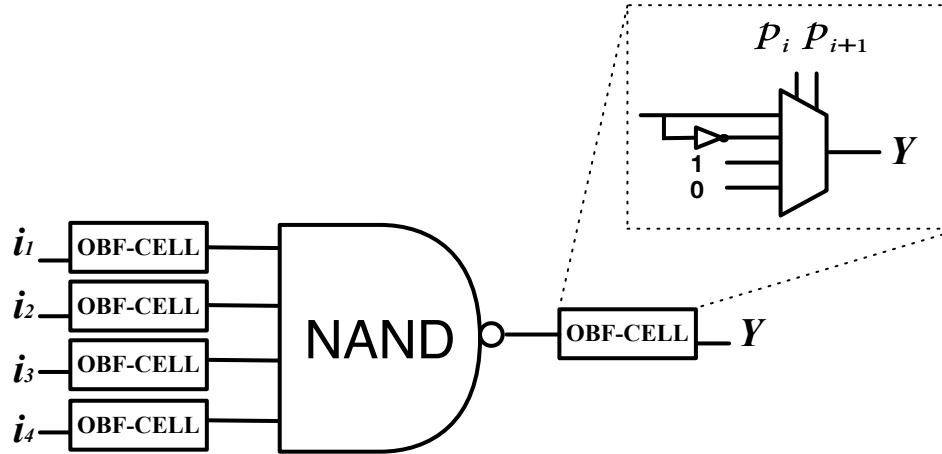
Figure 3.2: Schematic of a single *Obfusgate* that consists of 5 Obfuscells together with a 4-input NAND gate. This gate can have 162 different logic functions depending on the logical functions realized by each Obfuscell.

Figure 3.2 shows an Obfusgate comprising a NAND4 gate and five Obfuscells. Because an Obfuscell potentially has four different functions, I use a 4-to-1 multiplexer to model an Obfuscell and apply constant 0, constant 1, buffer and inverter to the multiplexer (Figure 3.2). The function of the camouflaged circuit can be resolved by finding appropriate values for the programming bits of the Obfuscells. For each NAND4-based Obfusgate, with five Obfuscells each having four possible functions, there are $4^5$ total configurations possible. Many of these $4^5$ configurations cause the Obfusgate to realize the same functions, and there are 162 unique 4-input logic functions that can be created by the NAND4 Obfusgate.

## 3.3 Transformable Interconnects

Different from the previous discussed two techniques, the principle of *transformable interconnects* is to provide connection obfuscation, which is proposed by Chen et al [4]. The implementation of this method is using two types of contacts in interconnection: magnesium (Mg) contacts which are conductors, and magnesium oxide (MgO) contacts

which are not conductors. During delaying, the Mg contacts will oxidize into MgO, so the attacker cannot distinguish which contacts were Mg.

An example of camouflaging using *transformable interconnect* is shown in Figure 3.3. The reverse engineer's view of the circuit is as shown in Fig. 3.3(a), and from this he will infer that d1, d2, and d3 cannot all be true wires. The configurations in Fig. 3.3(b), 3.3(c) and 3.3(d) represent the set of hypotheses for the true connectivity of the circuit. Each one of these would represent the circuit functionality under a single guess about which wire was a non-conducting dummy. The reverse engineer therefore models the transformable interconnect component as shown in Fig. 3.3(e), where the values of $p_0$ and $p_1$ select the true connectivity of the circuit. Now, just as in the previous components, the reverse engineer can use input-output examples from the circuit to infer the values of $p_0$ and $p_1$ and hence resolve the function of the circuit.
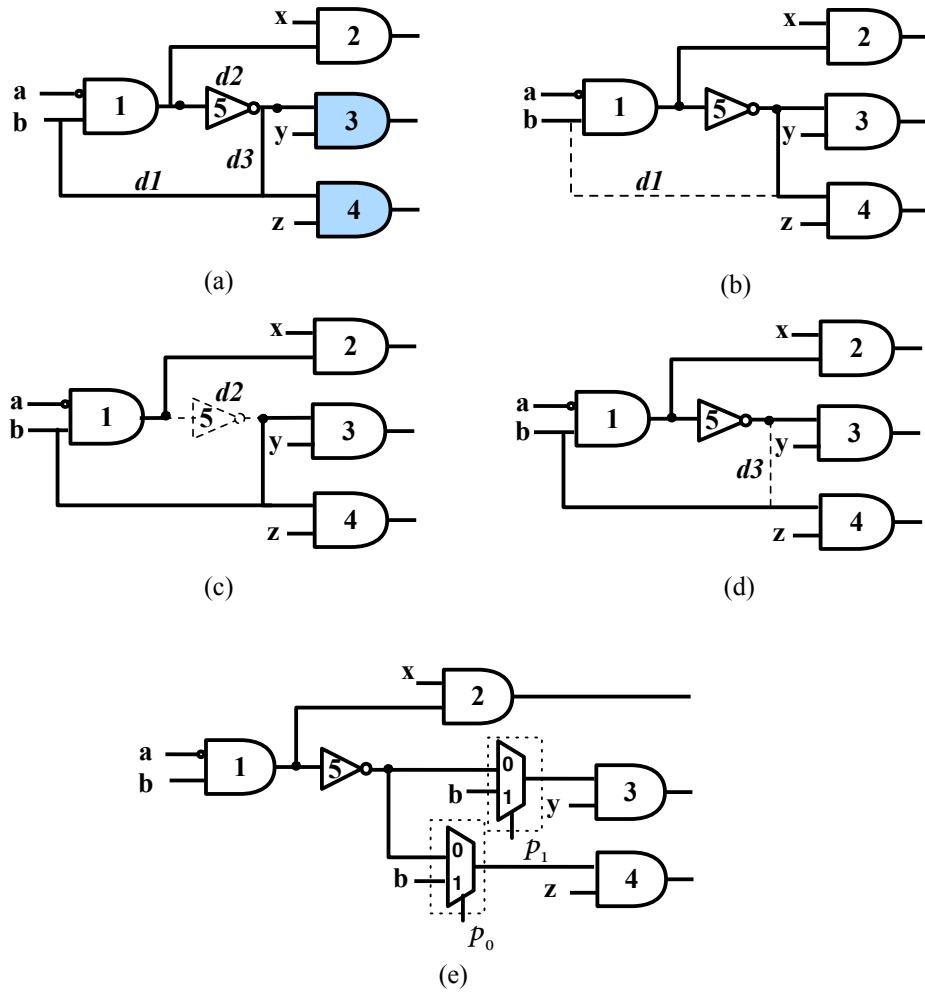
Figure 3.3: (a) Design as viewed by reverse engineers; (b, c, d) three valid circuit configurations when $d1$ or $d2$ or $d3$ is transformable interconnect; (e) attacker model of *transformable interconnect*.

# CHAPTER 4

# PROBLEM FORMULATION

---

**Algorithm 1** *Incremental SAT-based Deobfuscation:* Incrementally generate a set of constraints that are sufficient to identify the correct model of the circuit, and then solve the constraints to find the model.

---

1: $M(I, P, P') \leftarrow ckt(I, P, O) \wedge ckt(I, P', O') \wedge O \neq O'$      // see Fig. 4.1

2: $feas(P) \leftarrow \top$      // all programming vectors are feasible initially, unless the model itself imposes constraints

3: $feas(P') \leftarrow \top$

4: **for** $j = 1, 2, 3 \ldots$ **do**

5:      // number of satisfying assignments to $feas(P)$ is the number of programming vectors that remain feasible

6:      **if** $\exists I, P, P'$.    $\overbrace{M(I, P, P')}^{I \text{ distinguishes } P \text{ and } P'} \wedge \overbrace{feas(P) \wedge feas(P')}^{P \text{ and } P' \text{ both feasible}}$    **then**

7:          $\widehat{I}_j \leftarrow I$

8:          $\widehat{O}_j \leftarrow \text{QUERYORACLE}(\widehat{I}_j)$      // $(\widehat{I}_j, \widehat{O}_j)$ is the $j^{th}$ I/O pair discovered

9:          $feas(P) \leftarrow feas(P) \wedge ckt(\widehat{I}_j, P, \widehat{O}_j)$     // strengthen feasibility constraint on $P$ using new I/O pair

10:         $feas(P') \leftarrow feas(P') \wedge ckt(\widehat{I}_j, P', \widehat{O}_j)$     // strengthen feasibility constraint on $P'$ using new I/O pair

11:      **else**

12:          $\exists P. feas(P)$     // Find a single feasible programming assignment $P$

13:          **return** $P$

14:      **end if**

15: **end for**

---

From the previous chapter, I use multiplexer-based component to model the three techniques (see Figs. 3.1, 3.2 and 3.3). I now present a de-obfuscation algorithm to solve for the programming vector assignment that can configure the logic function of the model to be the same as the oracle circuit. The programming vector can thus reveal the actual connection and functionality of the camouflaged circuit. This algorithm is based on a series of input-output pairing; these pairs constrain which

programming vector values are feasible. A feasible programming vector is one that induces a circuit function that does not contradict any known input-output pairings.
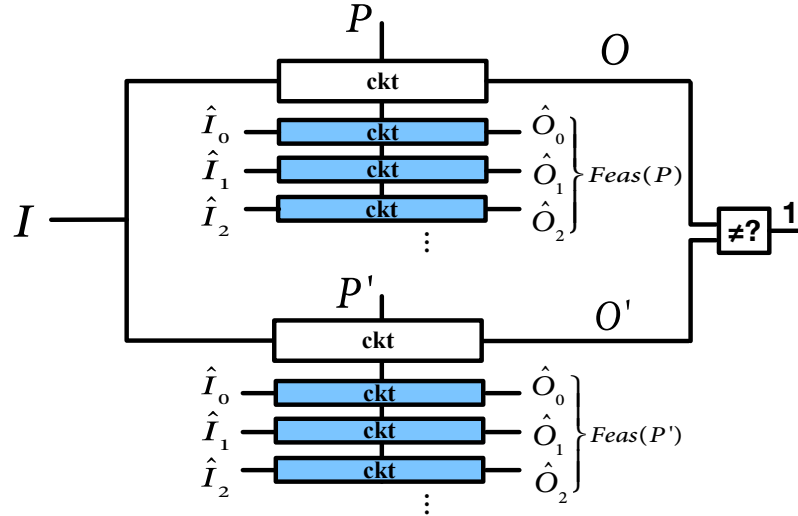


Figure 4.1: The unshaded components comprise a miter used to find conditions where two copies of the circuit produce different outputs due to different programming vectors. The shaded components enforce feasibility constraints that restrict programming vectors to be consistent with input-output examples.

## 4.1 Defining Notation

- Vector $I = \{i_0, \ldots, i_{m-1}\}$ represents an $m$-bit primary input vector to the circuit.

- Vector $P = \{p_0, p_1, \ldots\}$ represents a programming vector that specifies the logic function implemented by each camouflaged component in the circuit. The length of the programming vector depends on the number of camouflaged circuit elements and the number of possible realizations for each element. The value of $P$ together with the non-camouflaged circuit components together fully specify the logical function of the overall circuit. A programming vector is denoted as *feasible* if the logic function it induces does not contradict a set of known input-output examples. Learning new input-output examples incrementally constrains feasible values of the programming vector $P$.

13

- Vector $O = \{o_0, \ldots, o_{n-1}\}$ represents an n-bit primary output vector.

- The combinational circuit model, including all multiplexers and programming bits, is converted into a CNF formula $ckt$ using Tseitin encoding. I use $ckt(I, P, O)$ to denote the CNF formula of the circuit when $I$, $P$, and $O$ are the input variables, programming bits, and output variables respectively. Wherever $ckt(I, P, O)$ appears in Alg. 1 (at lines 1, 9, and 10), it always refers to a fresh copy of the circuit CNF with new variables for all internal circuit nodes. If two copies of the circuit CNF share a common input vector or programming vector, the respective inputs or programming vectors are equated to each other outside of the CNF of the combinational circuit.

- Subformula $M(I, P, P')$ in Alg. 1 (line 1) represents the CNF-encoded miter of two copies of the circuit, as shown by the unshaded blocks in Fig. 4.1. This formula is satisfiable if and only if there exists an input $I$, and programming vectors $P$ and $P'$ that cause the circuit to map $I$ to different output values. Restated, $M(I, P, P')$ is satisfiable if *some* $P$ and $P'$ cause the circuit to realize different logic functions. If $M(I, P, P')$ is unsatisfiable, then it means that *all* $P$ and $P'$ cause the circuit to realize the same logic function. On its own, this formula would typically be easily satisfiable and not meaningful, but it becomes useful when combined with additional feasibility constraints on $P$ and $P'$. In that scenario, the formula $M(I, P, P')$ is used for checking whether the constraints can be satisfied by two different realizable logic functions, or whether the constraints are strong enough that only a single realizable logic function satisfies them.

- Subformulas $feas(P)$ and $feas(P')$ denote the feasibility constraints applied to programming vectors $P$ and $P'$ respectively. These formulas are identical except for being applied to different copies of the programming vector. The formulas

evaluate to true only for the subset of programming vectors that are consistent with a set of input-output pairings obtained from the oracle. These feasibility constraints are CNF-encodings of the shaded blocks in Fig. 4.1.

The goal of the attacker is to recover the function of the obfuscated circuit by finding a value of $P$ that induces his model to realize the same function as the oracle. Two functions are equivalent if they produce the same outputs for all possible inputs, but equivalence is usually checked symbolically instead of by exhaustively applying inputs. However, symbolic equivalence checking cannot be applied between a model and a black-box oracle, so SAT-based reverse engineering relies on an oracle-guided synthesis approach, as described in the remainder of this paragraph. Given that there are programming vectors to select all possible functions of all camouflaged components, there necessarily exists one or more values of $P$ that will cause the model to realize the same function as the oracle. Because the attacker knows that there must exist such a value of $P$, he can find it by ruling out values of $P$ using input-output examples from the oracle until only a single function remains. At this point, the one function that is not ruled out is known to be equivalent to the oracle by the process of elimination. Any value of $P$ that induces the model to have this function is a solution to the deobfuscation problem.

## 4.2  SAT Solving of Camouflaged Circuit

Boolean satisfiability solving is a common way to reason circuit logic, and it is used widely in automated test pattern generation (ATPG) [18]. Using Tseitin encoding, a gate-by-gate translation can map the circuit logic into a (CNF)-encoded SAT problem. The number of variables in the CNF problem is equal to the number of nodes in the circuit, and the number of clauses in the CNF problem is linear in the number of circuit logic gates. If a CNF problem is satisfiable, a SAT solver can find an assignment of 0 or 1 to each variable such that the CNF problem is equal to 1.

The difference between ATPG and the de-obfuscation problem is that obfuscated components are represented by the CNF formula using the circuit constructs of Figs. 3.1, 3.2, and 3.3. Those three constructs are encoded into the CNF in the same way as the other nodes and gates. The variables in the CNF model also include programming vector, circuit inputs, and circuit outputs. The known input-output pairs are applied by using unit clauses to force an input or an output variable to be certain value. To avoid confusion with arbitrary input and output vectors ($I$ and $O$ respectively), an input-output pairing that is known to be correct as is denoted $\widehat{I}$ and $\widehat{O}$ with various subscripts.

## 4.3   Incremental-SAT Algorithm

Our oracle-guided incremental-SAT based algorithm is given in Alg. 1. Following the notation described at the start of this section, the algorithm uses the sub-formula $M(I, P, P')$ to check whether two programming vectors induce different logic functions, and uses $feas(P)$ and $feas(P')$ to constrain programming vector assignments to be consistent with all previously observed input-output pairs from the oracle. In Alg. 1, both $feas(P)$ and $feas(P')$ are typically initialized to $\top$ meaning that the programming vectors are initially unconstrained; however, when using the modeling construct in Fig. 3.1a where only the 00,01, and 10 values are used for each pair of programming bits to select one of three logic functions, the constraints are initialized to rule out the 11 assignment.

The feasibility constraints are increasingly strengthened as the algorithm iterates through the loop. At the $j^{th}$ loop iteration in Alg. 1, a satisfying assignment at line 6 produces an input vector $I$ that can distinguish two feasible programming vectors $P$ and $P'$. This vector $I$ is assigned to $\widehat{I}_j$ and the oracle is queried to obtain the corresponding output $\widehat{O}_j$. The pair $(\widehat{I}_j, \widehat{O}_j)$ is known to be a correct input-output pairing according to the oracle, and it is used to strengthen the programming vector

feasibility constraints at lines 9 and 10. To strengthen the constraints on $P$, a new copy of the circuit CNF formula is added with $P$ as its programming vector and $\widehat{I}_j$ and $\widehat{O}_j$ applied to inputs and outputs as unit clauses (line 9). The feasibility constraint on $P'$ is strengthened in the same way (line 10). The strengthening of the feasibility constraints corresponds to adding new shaded blocks in Fig. 4.1. Note that the strengthened feasibility constraints will necessarily have fewer solutions after being strengthened; specifically, among the values $P$ and $P'$ that satisfied the SAT formula at line 6, at least one will now be infeasible[1].

Once the feasibility constraints are sufficiently strong, there will no longer exist two different programming vectors that induce distinct logic functions while also satisfying the feasibility constraints. At this point, the SAT call at line 6 becomes unsatisfiable, and a final SAT call is made (line 12) to find a single programming vector $P$ that satisfies the feasibility constraints. Note that the value of $P$ that is discovered may not be a unique solution, but it is known that no other feasible $P'$ induces a different overall logic function[2], as this is necessary for the SAT call at line 6 to be unsatisfiable.

Relative to SAT-based attacks of El Massad et al. [24] and Subramanyan et al. [29], a distinguishing feature of our work is the use of incremental SAT. A typical SAT problem is encoded in CNF and solved by a SAT solver to output either a satisfying assignment or a result of *UNSAT* to indicate that no such assignment exists. In the process of solving the SAT problem instance, the solver spends considerable time learning from conflicts and making inferences to simplify the problem and guide its search toward a satisfying assignment. If solving a set of related SAT instances, it is desirable to reuse this reasoning to reduce the number of costly inferences made in

---

[1]Because $P$ and $P'$ induce different outputs under the input vector $\widehat{I}_j$, no more than one of them can induce output vector $\widehat{O}_j$, which is now known to be correct.

[2]If two programming vectors do not produce different outputs for any input vectors, then they induce the same logic function

each SAT call [8]; incremental SAT is the formulation that allows for efficient reuse of inference across related SAT instances. Our problem is amenable to solving by incremental SAT because each SAT query (line 6 of Alg. 1) is solving an instance obtained by adding clauses (at lines 9 and 10) to the previously-solved SAT problem instance. All inferences learned in one SAT problem are therefore still applicable in the subsequent one.

A number of engineering challenges are addressed in order to use an oracle-guided approach with incremental SAT solving. An overview is given here, with more information found in the user manual included with our program. The algorithm is implemented using a modified version of MiniSat [9] version 2.2.0. From within MiniSat, at each iteration of the algorithm, when a satisfying assignment to the CNF is produced (at line 6 of Alg. 1), the primary input values ($\widehat{I}_j$) are extracted and mapped into their corresponding signal names and printed to a file. The oracle, implemented as a standalone executable, is then queried (line 8) and the program waits for the oracle to map $\widehat{I}_j$ to $\widehat{O}_j$. Once the oracle has produced $\widehat{O}_j$, the program adds new clauses to the ongoing CNF problem in order to strengthen the constraints on $P$ and $P'$.

The MiniSat 2.2.0 (simp) version is used because it implements variable elimination and simplification before solving. The overhead cost of performing the simplifications is justified because the simplified constraints are carried forward and used in all future iterations. Because this version of MiniSat can eliminate variables, care must be taken to "freeze" certain variables so that they will not be eliminated. In our case, the programming vectors $P$ and $P'$, and the input vector $I$, are frozen. Being frozen means that variables will always remain in the SAT problem, and this makes it possible to read out their values whenever a satisfying assignment is found.
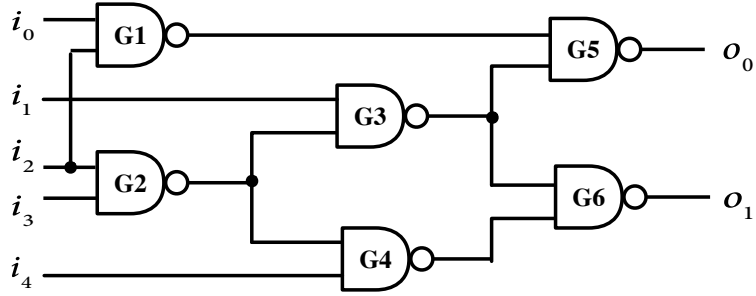
Figure 4.2: Gate-level netlist of circuit $c17$.

## 4.4 Baseline SAT-based De-obfuscation Algorithm

The baseline I used in my work is the approach that introduced by El Massad, Garg, and Tripunitara [24], which is similar to mine. Their algorithm can de-obfuscate a circuit in minutes while using brute force can take years [27]. Except for attacking different circuits, the major difference between my work and theirs is that I use incremental SAT to increase the performance. Unlike incremental SAT, the baseline treats each iteration as an unrelated SAT problem, so the solved variables and clauses in the previous iteration cannot be re-used in the current iteration. T runtime of the incremental approach is compared to the baseline in Chapter 5.

## 4.5 Illustrative

In this section, I use ISCAS-85 benchmark circuit c17 to demonstrate algorithm 1. The techniques shown here are NAND/NOR/XOR Camouflaged Standard Cells and fully camouflaged logic gates. In addition, I give an example to show why a successfully solved function does not agree with the oracle circuit gate-by-gate. Figure 4.2 is the gate level netlist for c17.

### 4.5.1 Example 1 - NAND/NOR/XOR Camouflaging

Figure 4.3 shows the camouflaged $c17$ model. I camouflage gate $G1$, $G4$, and $G5$. The camouflaged gate is equivalent to Figure 3.1(a). The program will find a

| j | num. feasible configurations | SAT solution (Alg. 1, line 6) | | | constraint learned | CNF statistics Incremental | | CNF statistics Baseline | |
|---|---|---|---|---|---|---|---|---|---|
| | | $\{I\}$ | $\{P\}$ | $\{P'\}$ | $(\hat{I}_j, \hat{O}_j)$ | num. vars | num. clauses | num. vars | num. clauses |
| 1 | 27 | 01000 | 10,01,00 | 00,10,00 | (01000,11) | 114 | 364 | 114 | 364 |
| 2 | 14 | 10100 | 00,10,01 | 00,01,01 | (00000,00) | 170 | 640 | 239 | 739 |
| 3 | 8 | 10110 | 00,00,01 | 00,01,01 | (10110,10) | 236 | 916 | 364 | 1114 |
| 4 | 6 | 00100 | 10,01,01 | 00,01,01 | (00100,00) | 294 | 790 | 489 | 1489 |
| 5 | 4 | 00000 | 01,01,01 | 00,01,01 | (00000,00) | 344 | 1066 | 614 | 1864 |
| 6 | 2 | 11101 | 01,01,01 | 01,01,00 | (11101,11) | 333 | 1342 | 739 | 2239 |
| 7 | 1 | UNSAT | | | - | 95 | 1614 | 864 | 2614 |

Table 4.1: Values produced at each iteration of Alg. 1 during deobfuscation of the camouflaged *c17* circuit in Figure 4.3. The number of feasible configurations is the number of programming vectors that satisfy the constraints at each iteration of the algorithm.

programming vector $P$ that can configure the camouflaged model to be functionally equivalent to the oracle. Because there are only three choices of function for to each camouflaged cell, two bits of the programming vector are used to represent a gate. The value of 00 selects a XOR gate, 01 selects a NAND gate, and 10 selects a NOR gate. The value of 11 is forbidden here by a constraint.

Table 4.1 recorded step-by-step results. In the table, $j$ is the iteration number; $I$, $P$, $P'$ are the assigned values in the solution to the SAT call at line 6 of the algorithm; $(\hat{I}_j, \hat{O}_j)$ are the input-output pairs obtained and used to strengthen feasibility constraints at lines 9 and 10 of the algorithm. For instance, the input value $I = 01000$ is found in the first iteration. Applying this input to the oracle produces an input-output pair (01000,11). In the next iteration, there are only 14 feasible configurations can satisfy this pair. Program then generates a different output when applying another input, which is the value of 10100. After six iterations, the program can not find such two different control vectors that can satisfy the previously found input-output pairs. At this time, the feasibility constraints imposed by these six pairs are sufficient to identify a unique logic function that matches the oracle and thus de-obfuscates the circuit. There is only one configuration remaining after 6 iterations. However, there can be more than one feasible configuration in the last step for some larger circuits. In Table 4.1, the number of clauses and unresolved variables is growing sub-linearly at each iteration for the incremental solver. This indicates that the solver is making

20

simplifications to the problem as it runs, which is important for large camouflaged designs.
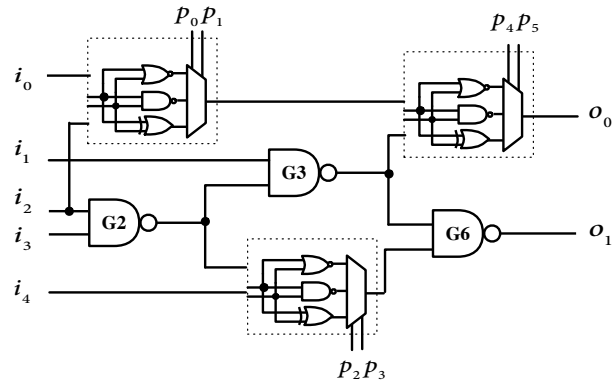


Figure 4.3: Modeling of $c17$ benchmark with G1, G4, G5 camouflaged using NAND/NOR/XOR camouflaging.
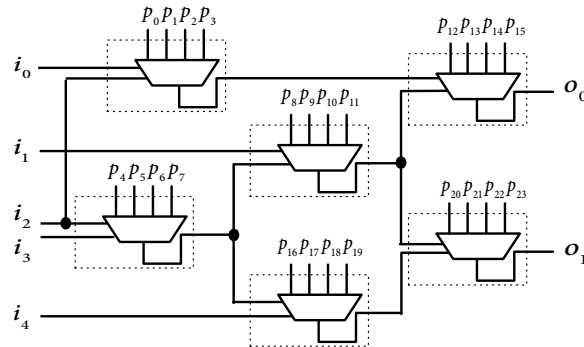


Figure 4.4: Modeling of $c17$ with all gates fully camouflaged. In this scenario, the reverse engineer only knows the routing.

### 4.5.2   Example 2 - All Gates are Camouflaged

In this scenario, all the logic gates in circuit c17 are fully camouflaged, the only information left is the routing. The model is shown in Fig. 4.4 which is based on $c17$, and the solution is shown in Fig. 4.5. Though different from the oracle on a gate-by-gate basis, the resolved function is equivalent to the original. Solving a circuit

with all gates are camouflaged is very hard, the program cannot de-obfuscate c432 after three days when all 160 gates are camouflaged.[3]
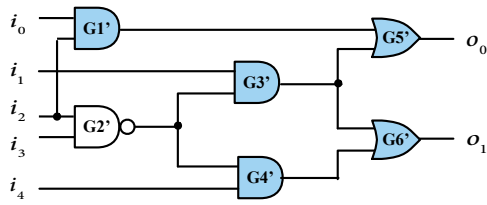


Figure 4.5: Resolved function of $c17$ when all gates are fully camouflaged.

---

[3]The ISCAS-85 benchmark $c432$ has 160 gates, some of which have more than 2 inputs. Because I use 2-input obfuscated gates, I map $c432$ into a circuit with 209 gates of 2 or fewer inputs, and then obfuscate all 209 cells of the remapped circuit.

# CHAPTER 5

# IMPLEMENTATION AND USER GUIDE

## 5.1 Introduction to Software

### 5.1.1 Purpose

The Oracle-Guided Incremental SAT Solver (Solver) is designed to solve *camouflaged* circuit with extremely high efficiency. The *camouflaged* circuit model represents the reverse engineer's uncertainty about the logic gates. The *oracle* represents the real physical object, where the user has only the ability to apply inputs and observe outputs, but cannot look inside to see what the gates are. The *Solver* will use the *oracle* as guide to solve for the logic function of the *camouflaged* circuit.

### 5.1.2 Principle

Solver executes a loop that continually finds new input and output vectors using SAT queries and an oracle circuit model. After some number of iterations, the constraints accumulated are sufficient to rule out all logical functions except for the one that is the true function of the obfuscated circuit.

### 5.1.3 Terminology

The additional terminology used in this chapter is listed below:

- Oracle program: an executable program that can produce the correct circuit output for any input. The input is provided to it via file PI.txt, and the corresponding output is written to PO.txt.

- Camouflaged circuit: obfuscated *oracle* circuit.

- Allowed values: Allowed values are the sets of values that the programming bits for each obfuscated component can take. Given that the programming bits select the functionality of the circuit, the combinations of the allowed values for all components represents the space of hypotheses for the overall circuit function. Finding specific values from these choices is the deobfuscation problem, and is the goal of our program.

## 5.2  Installation Tutorial

### 5.2.1  Dependencies

*NOTE:* The *Solver* is based on MINISAT.

- MINISAT module: modified from original MINISAT

  - (MROOT)/core: includes MINISAT solver head file, implementation file and related supporting file.

  - (MROOT)/mtl: includes MINISAT templates and make file.

  - (MROOT)/utils: includes MINISAT utilities.

- Solver module: solver module

  - (MROOT)/simp: includes *Solver* main file and MINISAT SimpSolver source file.

  - (MROOT)/incre: includes *Solver* source file and all the related utilities.

  - (MROOT)/Oracle: includes sample *Oracle* and sample Shell script.

### 5.2.2  Installation

Makefile is included in (MROOT)/simp directory, change to MROOT directory and use command below to install

1. *$ export MROOT=(solver-dir)*

2. *$ cd simp*

3. *$ make rs*

4. *$ cp SOLVER_static (install-dir)/SOLVER*

**NOTE:**

1. The minimum requirement for complier is **g++ 4.9**.

2. If make fails, use **$ make clean** and try again.

### 5.2.3   Command Line Usage

After installation, *solver* can be accessed from command line:

$ SOLVER [options] $<$Cam.$v >$ $<$Orac.$sh >$

- Cam.v : the Verilog netlist of the circuit to be de-obfuscated. The netlist must have the necessary annotations for the PIs and annotations to define the allowed values for the programming bits. Note that only a restricted subset of Verilog can be used, as defined below.

- Orac.sh: shell script that the solver will execute to query *Oracle*.

- -d, - -debug: change to debug mode, solver will generate log message and log files.

- -o, - -outfile: export solution to this file .

For example, if the *oracle* shell is "c432-Oracle.sh" and the *camouflaged* model is "c432-mux4-101.v", then the command can be:

$ SOLVER -d - -outfile Solution.txt c432-mux4-101.v c432-Oracle.sh

### 5.2.4  Description of oracle program

During the solving, *Solver* will repeatedly query the oracle by generating a 'PI.txt' file and reading a 'PO.txt' file. For stability sake, these filenames can not be changed. Please make sure your oracle program will can read 'PI.txt' and export 'PO.txt' in working directory. Sample PO.txt is in folder *Oracle*. The first line are the signal names, and the second line are the signal values. Each net name or value in PO file is seperated by a tab, each line in PO file is seperated by a line break. PI file uses the same format. For example:

```
N1(\t)N2(\t)N3(\t)N4(\t)N5(\t)CONST1(\t)CONST0(\n)
1(\t)1(\t)1(\t)0(\t)1(\t)1(\t)0(\n)
```

The user's oracle program can produce the outputs by any means they desire. In a reverse engineering setting, the oracle program could be a script that physically queries the obfuscated circuit via its scan chain. For sake of evaluation, users may wish to implement an oracle program that run simulation of a non-obfuscated circuit function, or queries a pre-programmed exhaustive look-up table of PI-PO pairs. In the example oracle program we provide, the PIs are mapped to POs by evaluating the circuit CNF using Minisat with appropriate inputs applied.

### 5.2.5  Format of camouflage circuit model

The camouflaged circuit model, including all programming bits is written in a limited subset of gate-level Verilog, the following is an example:

```
module  c17 (N1,N2,N3,N4,N5,N10,N11,CONST1,CONST0,D_0,D_1,D_2,D_3);
input  N1,N2,N3,N4,N5,CONST1,CONST0 ;//RE__PI;
input D_0,D_1 ;//RE__ALLOW(00,01,10,11);
input D_2,D_3 ;//RE__ALLOW(00,01,10,11);
output N10,N11;
wire N6,N7,N8,N9,D_0_NOT,D_1_NOT,N7_NOT,N7_OBF,ED_0,ED_1,ED_2,ED_3,ED_4,
    ED_5,ED_6,ED_7,ED_8,ED_9,D_2_NOT,D_3_NOT,N6_NOT,N6_OBF,ED_10,ED_11,
    ED_12,ED_13,ED_14,ED_15,ED_16,ED_17,ED_18,ED_19;
nand2 gate1( .a(N1), .b(N3), .O(N6) );
nand2 gate2( .a(N3), .b(N4), .O(N8) );
nand2 gate3( .a(N2), .b(N8), .O(N7) );
```

```
nand2 gate4( .a(N8), .b(N5), .O(N9) );
nand2 gate5( .a(N6_OBF), .b(N7), .O(N10) );
nand2 gate6( .a(N7_OBF), .b(N9), .O(N11) );
inv1 gate7( .a(D_0), .O(D_0_NOT) );
inv1 gate8( .a(D_1), .O(D_1_NOT) );
inv1 gate9( .a(N7), .O(N7_NOT) );
and2 gate10( .a(N7), .b(D_0_NOT), .O(ED_0) );
and2 gate11( .a(N7_NOT), .b(D_0_NOT), .O(ED_1) );
and2 gate12( .a(CONST1), .b(D_0), .O(ED_2) );
and2 gate13( .a(CONST0), .b(D_0), .O(ED_3) );
and2 gate14( .a(ED_0), .b(D_1_NOT), .O(ED_9) );
and2 gate15( .a(ED_1), .b(D_1), .O(ED_7) );
and2 gate16( .a(ED_2), .b(D_1_NOT), .O(ED_5) );
and2 gate17( .a(ED_3), .b(D_1), .O(ED_4) );
or2  gate18( .a(ED_4), .b(ED_5), .O(ED_6) );
or2  gate19( .a(ED_6), .b(ED_7), .O(ED_8) );
or2  gate20( .a(ED_9), .b(ED_8), .O(N7_OBF) );
inv1 gate21( .a(D_2), .O(D_2_NOT) );
inv1 gate22( .a(D_3), .O(D_3_NOT) );
inv1 gate23( .a(N6), .O(N6_NOT) );
and2 gate24( .a(N6), .b(D_2_NOT), .O(ED_10) );
and2 gate25( .a(N6_NOT), .b(D_2_NOT), .O(ED_11) );
and2 gate26( .a(CONST1), .b(D_2), .O(ED_12) );
and2 gate27( .a(CONST0), .b(D_2), .O(ED_13) );
and2 gate28( .a(ED_10), .b(D_3_NOT), .O(ED_19) );
and2 gate29( .a(ED_11), .b(D_3), .O(ED_17) );
and2 gate30( .a(ED_12), .b(D_3_NOT), .O(ED_15) );
and2 gate31( .a(ED_13), .b(D_3), .O(ED_14) );
or2  gate32( .a(ED_14), .b(ED_15), .O(ED_16) );
or2  gate33( .a(ED_16), .b(ED_17), .O(ED_18) );
or2  gate34( .a(ED_19), .b(ED_18), .O(N6_OBF) );
endmodule
```

There are several points that need attention regarding the Verilog netlist:

- The line declaring primary inputs should be followed by "//RE__PI;".

- The line declaring control bits should be noted by "//RE__ALLOW();".

- Allowed values for each group of control bits should be written inside parentheses of "//RE__ALLOW();", for example "//RE__ALLOW(1,0);". If one camouflaged gate requires more than one control bits, for example it needs two bits, use the format "//RE__ALLOW(00,01,10,11);".

- The number of control bits in an input line must be equal to the length of allowed values after it. For example, "input D_1; //RE__ALLOW(00,01,10,11);"

is illegal because it defines a single programming bit but describes a set of 2-bit values for the bit to take.

- Primary inputs named CONST1 and CONST0 will be interpreted as values 1 and 0.

- Solver can accept the following gate types:

    - inv

    - and (with any number of fanin)

    - or (with any number of fanin)

    - xor

    - nor (with any number of fanin)

    - nand (with any number of fanin)

    - buf

## 5.2.6  Flow Diagram

```
                    ┌──────────────────┐
                    │   translate and  │
                    │    build miter   │
                    └──────────────────┘
                             │
                             ▼
┌──────────────┐     ┌──────────────────┐
│  instantiate │     │     solve to     │
│  new copy of │     │    find a PI to  │
│  camouflaged │ ──▶ │   differentiate  │
│  circuit and │     │   two feasible   │
│  assign newly│     │    programming   │
│  found PIPO  │     │      vectors     │
└──────────────┘     └──────────────────┘
       ▲                     │
       │                     ▼
┌──────────────┐        ◇─────────◇           ┌──────────────────┐
│  find corre- │  Yes  │ solvable? │    No     │  based on col-   │
│  sponding PO │ ◀──── │           │ ────────▶ │  lected PIPOs    │
└──────────────┘        ◇─────────◇           │  solve for final │
                                              │   programming    │
                                              │   vectors that   │
                                              │     deobfus-     │
                                              │   cates circuit  │
                                              └──────────────────┘
```
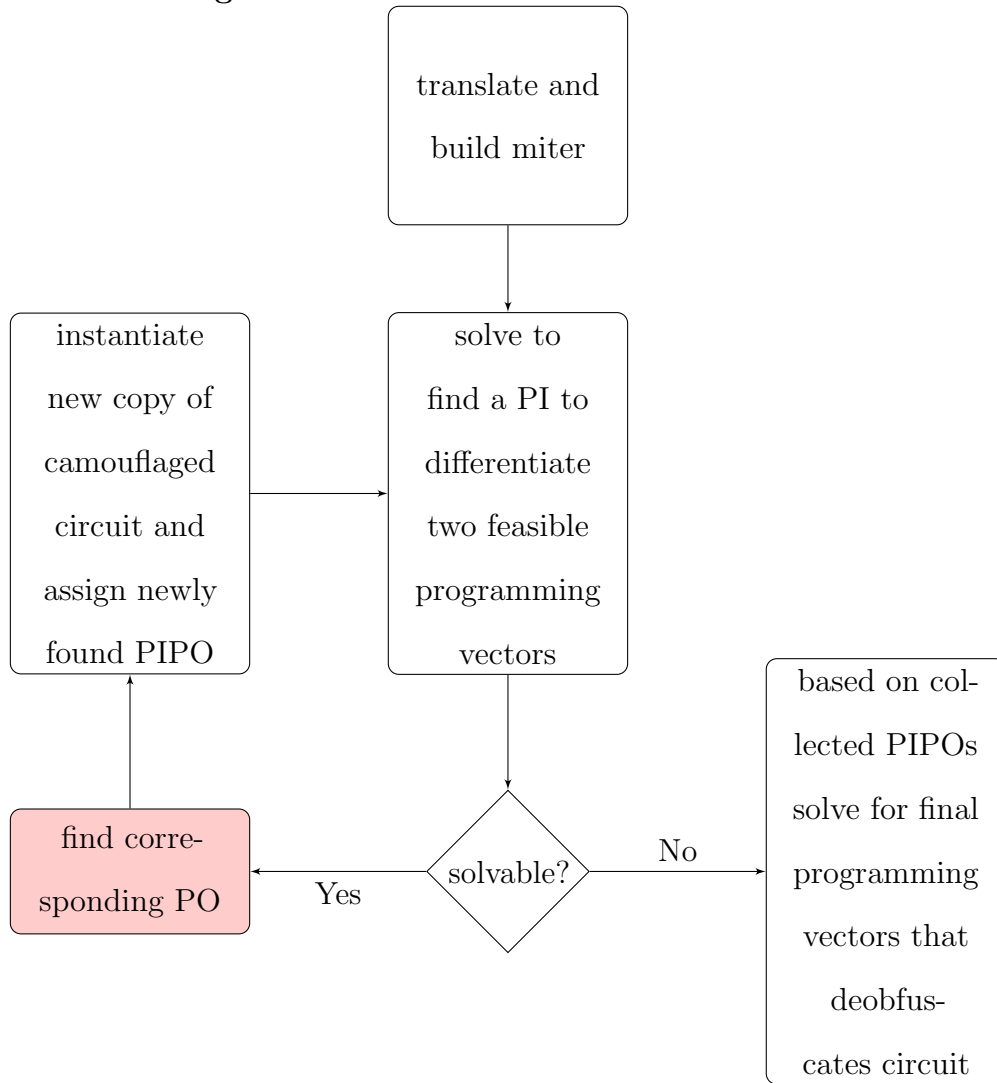
**NOTE:** The white filled blocks are solver tasks. And the red filled block belongs to user defined oracle.

## 5.3  Architecture and implementation details

The real computation is based on encoded-CNF. So principle of the software is very close to a compiler. The front end involves syntax analyzer and symbol table, and back end is responsible for generating intermediate representation, which is the encoded-CNF model. Different from the traditional concept of compiler, my software doesn't include code optimization and code generation module. It calls MINISAT's

interface to convert the Tseitin-encoded CNF model into the internal data structure and solve. The communication between MINISAT and compiler happens in each iteration.The compiler provides new encoded-CNF clauses for the circuit, and MINSAT provide the intermediate result indicating whether the problem is solved. In this section, the architecture, class structure, class interface and implementation, and the verification tool are explained.

### 5.3.1 Control Flow

Based on the flow diagram provided in the previous section, the main function is written as shown:

```
  MiterSolver *MTR = new MiterSolver;
  MTR->buildmiter();
  delete MTR;

  while(1)
  {
      AddonSolver *ADD = new AddonSolver;

      ADD->start_solving();
      if(IncreSolver::check_ret() == l_True)
      {
        ADD->queryOrac();
        ADD->continue_solving();
      }
      else
      {
          break;
      }
          delete ADD;
  }

  SoluFinder *finder = new SoluFinder;
  finder->find_solu();
  IncreSolver::print_state();
  finder->print_solution();
  delete finder;
  return 0;
```

There are some challenges needed to solve when writing this program.

1. Working as an API, this software must be encapsulated and extensible.

30
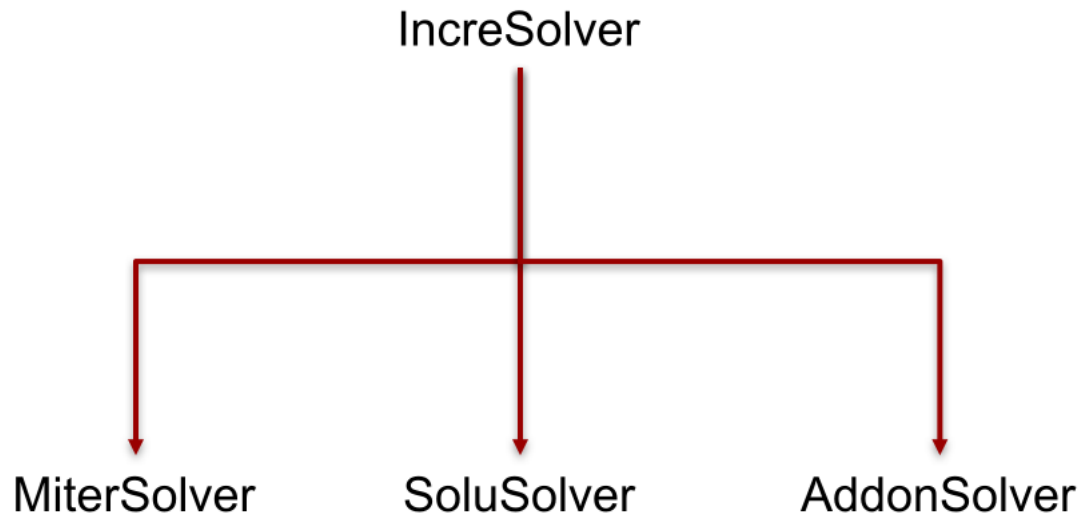
Figure 5.1: class structure of the software.

2. Considering that some large design require many iteration before completing, memory usage is critical, so this software is running iteratively rather than recursively. However, the intermediate result will be removed after each iteration.

3. For the convenience of debugging, the status while running should be accessible for inspection.

### 5.3.2 Class Structure

To deal with the first challenge, this software is wrapped by base class $IncreSolver$. The class structure is shown in Fig. 5.1.?

The class $MiterSolver$ works as the compiler, its responsibility is to translate the input Verilog code into Tseitin encoded-CNF model. It also duplicates two copies of the camouflaged circuit and builds the miter, which is the line 1, 2, 3 in Alg. 1. Class $AddonSolver$ solves the grown CNF model and applies the most recent input to the $oracle$ in order to get the corresponding output. The last step belongs to class $SoluFinder$, it duplicates the camouflaged circuit CNF as the number of input-output

pair, uses MINISAT's interface to solve and collects the solution to the programming vector that de-obfuscates the circuit.

To prevent the intermediate result from being deleted, all the necessary internal variables and data structure are stored as *static*. So even though the instance of *AddonSolver* is deleted, the intermediate data and the grown CNF model can be kept. And the *static* variables can also be accessed from the base class regardless of whether there is an instance of *IncreSolver*. But using static variables requires there should be only one instance of class. By disabling the copy constructor, this software is a typical singleton design.

### 5.3.3 IncreSolver

As the base class, the role of *IncreSolver* is mainly to declare and defines global member variables and virtual functions. However, this class also provides some common utilities. The global variables are declared as below, with description of each one gives in comments:

```
public:
    static Minisat::lbool ret;                              //
        indicator: indicate whether this iteration in addon is sat or
        not

    std::map<int, std::string> Solution;                   //
        container: store final Solution

    std::map<int, std::string> PItemp;                     //
        container: store temporary (only in this iteration) miter PI
        index->value
    std::map<int, std::string> POtemp;                     //
        container: store temporary (only in this iteration) oracle PO
        index->value

    static std::vector<std::map<int, std::string> > OracPIs;    //
        container: store all temp PIs
    static std::vector<std::map<int, std::string> > OracPOs;    //
        container: store all temp POs

    IncreSolver();
    ~IncreSolver();

    static Minisat::lbool check_ret();                     // tools
        : check ret before any instanization
```

```cpp
    static void print_state();                              // tools
        : print info including CPU, memory, time ,iterations

protected:

    static bool debug;                              // indicator: level of
        verb
    static bool out_file;                           // indicator: exist
        solution file or not
    static bool time_limit;                         // indicator: set time
        limited or not

    static int niter;                               // indicator: number of
        iterations
    static int time_bound;                          // value: time limited

    static const char * Came_file_path;             // path: input
        Camouflage file path
    static const char * Orac_file_path;             // path: input Oracle
        SHELL file path
    static const char * target_cnf;                 // path: output of
        buildmiter, input of solver, and output of addon
    static const char * Solver_solution;            // path: final solution
        path

    static std::vector<int> camPIndex;              // container:
        miter first circuit's PI, and also it the oracle's PI
    static std::vector<int> camPOindex;             // container: PO
        index list
    static std::vector<int> camCBindex;             // container: CB
        except duplicated circuit
    static std::vector<int> miterCBindex;           // container: CB
        include duplicated circuit
    static std::vector<int> camCB2index;            // container:
        duplication's CB
    static std::vector<int> nodes2grab;             // container:
        variable need to be frozen during incremental solving

    static std::map<int, std::string> indexVarDict;    // map: store
        map of index to netname
    static std::map<std::string, int> varIndexDict;    // map: store
        map of netname to index

    std::map<int, std::string> CB1temp;                // map: store
        temporary (only in this iteration) original CB index—>value
    std::map<int, std::string> CB2temp;                // map: store
        temporary (only in this iteration) duplication CB index—>value

    std::vector<int> addon_CB1;                     // container: store
        temporary (only in this iteration) first duplication CB index
    std::vector<int> addon_CB2;                     // container: store
        temporary (only in this iteration) second duplication CB index
    std::vector<int> addon_PI1;                     // container: store
        temporary (only in this iteration) first duplication CB index
```

33

```cpp
    std::vector<int> addon_PI2;                          // container: store
        temporary (only in this iteration) second duplication PI index
    std::vector<int> addon_PO1;                          // container: store
        temporary (only in this iteration) first duplication PO index
    std::vector<int> addon_PO2;                          // container: store
        temporary (only in this iteration) second duplication PO index

    static int cktTotVarNum;                             // values: number of
        wire including miter and oracle circuit
    static int camVarNum;                                // values: total number
        of wires + inputs + CBs + outputs in the original cam ckt
    static int miterOutIndex;                            // values: last index of
         miter

    static std::vector<std::string> camCNFile;        // CNF: original
         Camouflaged circuit CNF

    static progress_t bar;                               // indicator:
        process bar
    static Minisat::SimpSolver S;                        // object: used
        for solve add on
    static Minisat::SimpSolver S_final;                  // object: used
        for solve finalSolue

    static clock_t start;                              // indicator: starting
        time
    static clock_t totoal_all;                         // indicator: all thread
         total time
    static clock_t total_sub;                          // indicator: sub—thread
         total time
```

### 5.3.4 MiterSolver

Class *MiterSolver* is responsible for parsing the input netlist and building the miter. The complete class definition is shown as below:

```cpp
  class MiterSolver : public IncreSolver
{

private:
     MiterSolver(MiterSolver&);
    int baseMtrVarNum;        // values: total variable number (original
        + duplicated + XOR + OR)

    std::vector<std::string> baseCnfMtrLs;           // CNF: completed
        miter (including original Cam, duplicated Cam, XOR, Or)
    std::vector<std::vector<int> > inputs;           // container: same
        to inputs (includes PI and CBs)
    std::vector<std::string> forbidden_string;       // CNF: forbidden
        string
```

```
public:
    MiterSolver();              // constructor: initialize base class and
        milterSolver
    ~MiterSolver();             // deconstructor
    void buildmiter();          // main: build CNF formatted miter and
        export to Miter_file_path
private:
    void genOracCNF(char const * OracPath, int start);      // main:
        parse "OracPath", generate CNF, index start on "start"
    void genCameCNF(char const * CamePath);                 // main: parse
        "CamePath" and generate CNF
    std::vector<std::string> forbidden_bits(std::string line, std::::
        vector<int> target);    // main: process forbidden options
    std::vector<std::string> connectPO_xor(std::vector<int> &posIndex,
        int &camVarNum, int &xorInt);       // tools: connect POs using
        xor, used only for two duplicated circuit
    void formatCheck(std::vector<std::string> netlist);
};
```

Note that as a singleton design, the copy constructor is disabled. Except for the copy control functions, the only public member function is *buildmiter*(). This function is the flow control function is this class, which can be described as Fig. 5.2. The function *forbidden_bits* is responsible for the conversion from allowed bits to forbidden bits. The allowed bits defined in input Verilog will be translated in this function and added to the CNF model as constrain to prevent any non-allowed values from being assigned to those programming bits.

### 5.3.5   AddonSolver

Class *AddonSolver* is the most important part. Its job can be roughly divided into three parts:

1. solve CNF model generated by *MiterSolver* or the previous iteration.

2. apply input to the oracle and generate the corresponding output.

3. strengthen the current model by adding constraints for the new input-output vector

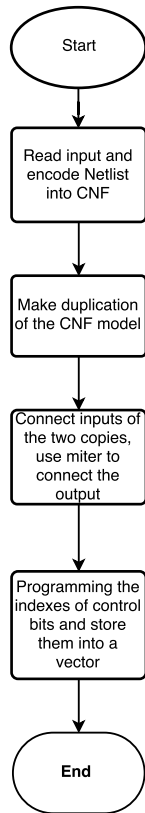The detailed work flow is described in Fig. 5.3. And the complete class definition is shown below:
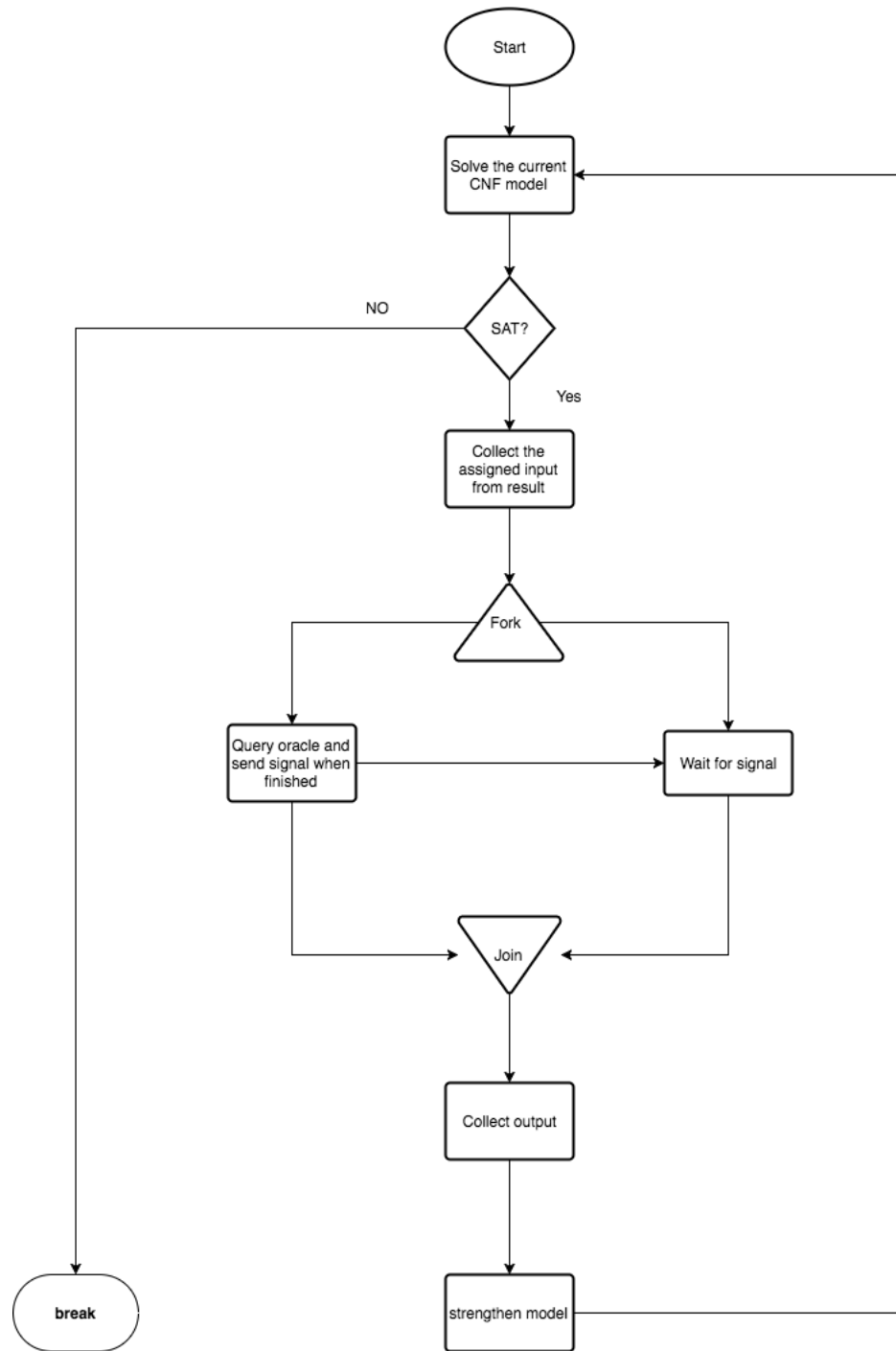
Figure 5.2: class MiterSolver's work flow

Figure 5.3: class AddonSolver's work flow

```
  class AddonSolver: public IncreSolver
{

public:
    AddonSolver();
    ~AddonSolver();
    void start_solving();
    void continue_solving();
    void queryOrac();

private:
    void print_solution(const char * path);
    void freeze();
    void print_map(std::map<int,std::string> &container, std::ofstream &
        outfile);
    void solve();
    void export_PI();
    void parse_PO();
    void run_shell();

};
```

The function *run_shell()* is the implementation of querying oracle. It based on C standard library function *system(char\* command)*. By calling this function, the current process is forked into two identical processes. One of the process calls the argument *command*, while the other process is stalled until the previous process sends a exit signal. The string *command* is a shell command.

Function *continue_solving()* is designed to collect output generated from oracle and add more constraints to the current model. To be specific, it duplicates two more encoded camouflaged circuit copies and connects the control bits to the original two copies respectively, then the newly found input-output pair is assigned to both of them as constraints.

Due to the incremental solving, some unnecessary variables will be eliminated. However, some important variables, such as control bits, input bits and output bits, can also be deleted. So I write function *freeze()* to notify the SAT solver of which variables should be frozen.

### 5.3.6 SoluFinder

As the last class, *SoluFinder* is responsible to use all the collected input-output pairs to create a CNF model, then solve it and translate the solution into human readable form. The form indicates the value of the programming bits, which can in turn specify the actual functionality of the camouflaged gates. The work flow can be summarized as Fig. 5.4 and the class definition is shown below. Similar to class *AddonSolver*, the function *freeze()* can freeze those critical variables and prevent them from be eliminated by solver.

```cpp
class SoluFinder : public IncreSolver
{

public:
    SoluFinder();
    ~SoluFinder();
    void find_solu();
    void print_solution();

private:
    int num2dup = 0;
    int totVarNum = 0;
    int clauseNum = 0;

    std::vector<std::string> finalCNF;

    void solve_it();
    void freeze();
};
```
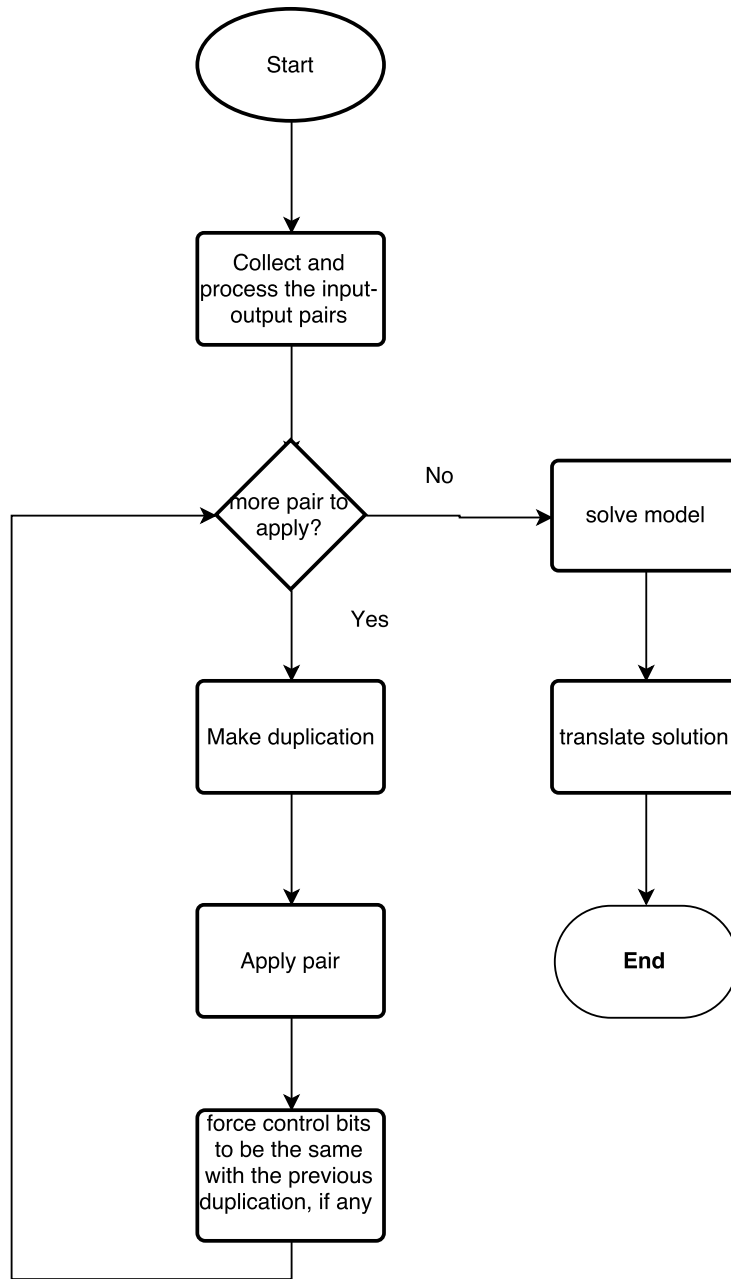
Figure 5.4: class SoluFinder's work flow

# CHAPTER 6

# EVALUATION OF DE-OBFUSCATION ALGORITHM

I evaluate my algorithm using a set of ISCAS-85 combinational benchmarks [11]. I also investigate the way to utilize both functionality and connection obfuscation. The results presented in this chapter are based on the attacker model introduced in chapter 3.

## 6.1 Evaluation of Camouflaging Techniques

I apply my incremental deobfuscation algorithm to reverse engineer designs camouflaged using *Camouflaged Standard Cells* [27], *Obfusgates* [23], and *Transformable Interconnects* [4]. I implement each of the three camouflaging techniques randomly as summarized in Tab. 6.1 and described here:

- For *Camouflaged Standard Cells*, I randomly choose gates from NAND2, NOR2, and XOR2 to camouflage. Those gates are the types that can be realized by camouflaged standard cells.

- For *Obfusgates*, although each NAND4 obfusgate can implement 162 different logical functions, most of the functions do not exist in the ISCAS-85 benchmarks. The gates from the ISCAS benchmarks that can be realized by the NAND4 obfusgates are as shown in Tab. 6.1.

- For *Transformable Interconnects*, I randomly select a wire to obfuscate and choose three wires driven by other gates' output as the input to the logic model. But before this, I levelize circuit based on dependency, and all the PI are assigned
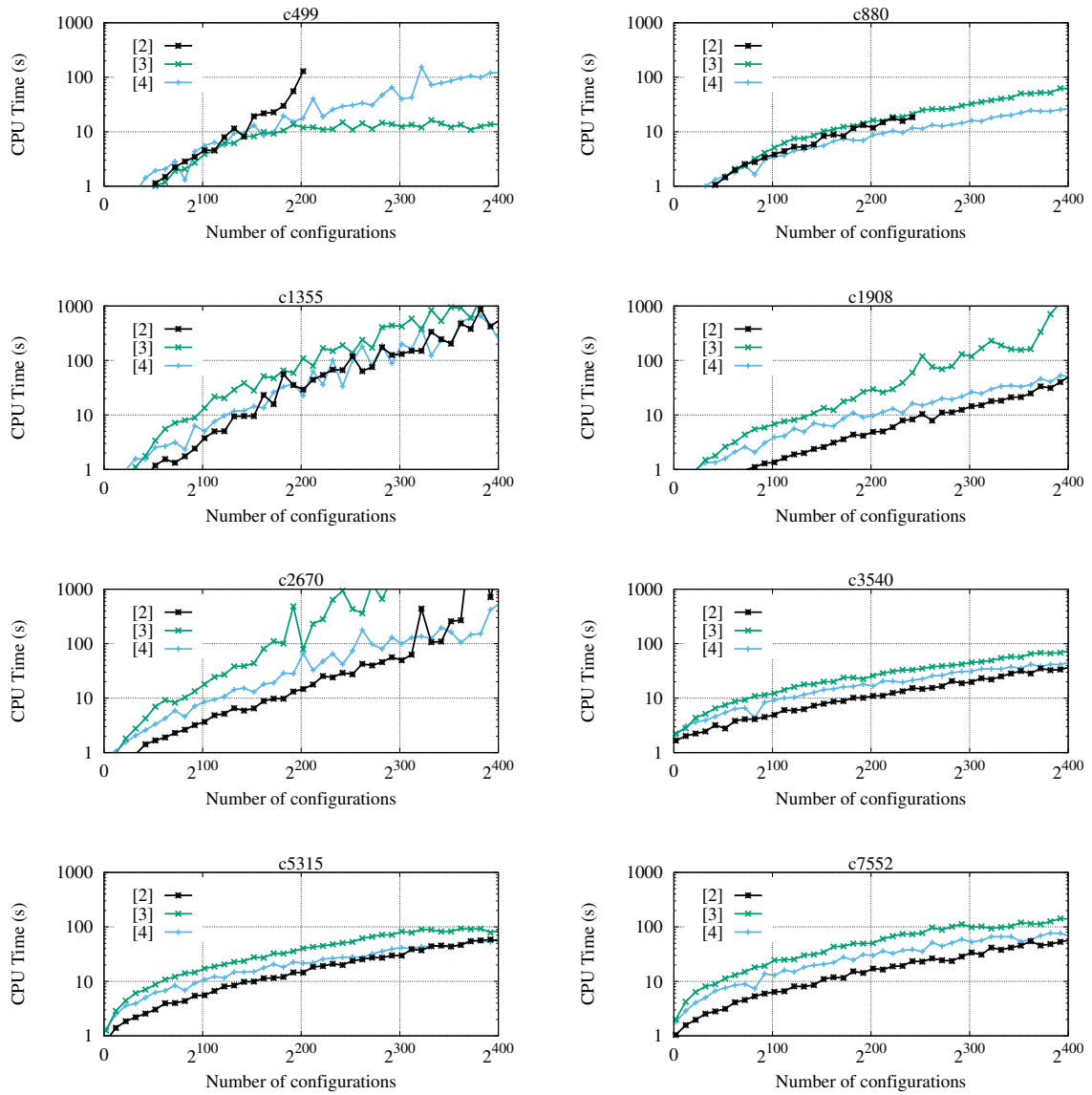
Figure 6.1: Plots show average runtime to de-obfuscate eight ISCAS-85 benchmarks with varied numbers of randomly obfuscated components using the camouflaging techniques presented in *Camouflaged Standard Cells*[27], *Obfusgates*[23] and *Transformable Interconnects*[4]. Specifically, the runtimes shown are the average runtimes over 10 random trials for each technique and number of obfuscated components.

as level 1. For example. a gate has inputs from PI and another gate at level 2, then this gate is in level 3. Note an attacker may be able to identify a loop and know it is dummy wire, so I only use dummy wire from the levels close to primary input.

I apply all three techniques to the eight ISCAS benchmarks, as shown in Fig. 6.1. In each case, I vary the number of components that are obfuscated, and for each number of obfuscated components, I repeat the experiment 10 times making different random choices of which components to obfuscate, and plot the average runtime. To provide a common framework for comparison, I plot the de-obfuscation runtime against the number of possible configurations in Fig. 6.1. The number of possible configurations is $2^x$ where $x$ is the number of programming bits needed to select the functionality of the circuit. The value of $x$ also indicates the number of camouflaged cells in the circuits. Specifically, in Fig. 6.1, the numbers of camouflaged cells for *Camouflaged Standard Cells* [27], *Obfusgates* [23], and *Transformable Interconnects* [4] are $x/2$, $x/5$, and $x/2$, respectively. Note that I don't consider here that some programming bits select the same logic function.
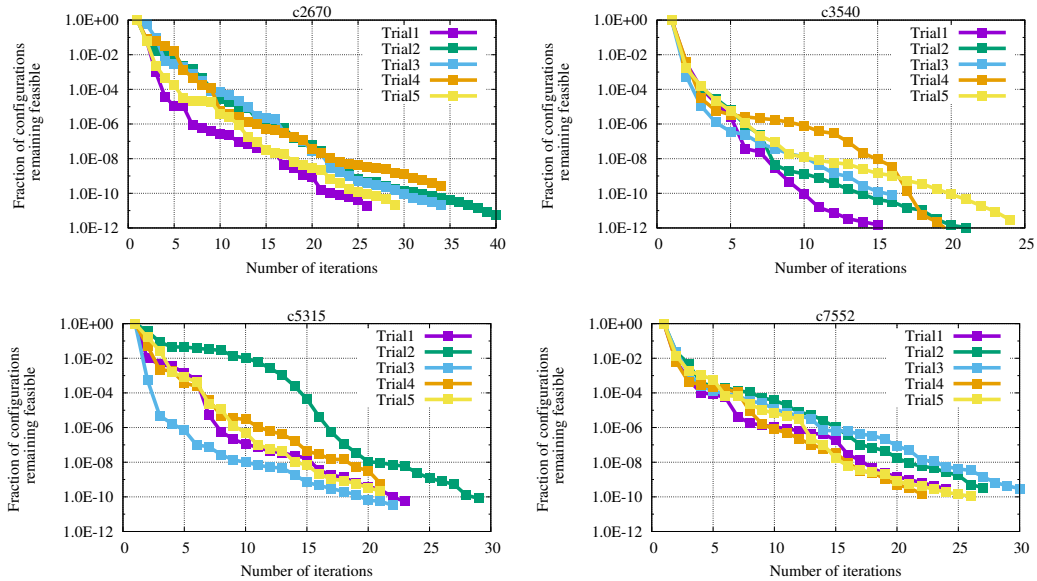


Figure 6.2: Eliminating feasible configurations using input-output examples generated by the de-obfuscation algorithm for circuits c2670, c3540, c5315, and c7552 with 51 gates camouflaged using camouflaged standard cells that can implement NAND, NOR or XOR gates. The initial model of each camouflaged circuit has $3^{51}$ configurations. The five trials in each plot denote five different random choices of which gates to camouflage.

| Camoflaging Technique | Camouflagable components in ISCAS-85 benchmarks |
|---|---|
| NAND/NOR/XOR camouflaged cells [27] | NAND2, NOR2, XOR2 |
| NAND4 Obfusgates [23] | AND/NAND(2,3,4), INV, OR/NOR(2,3,4), BUFFER |
| Tranformable Interconnects [4] | any net |

Table 6.1: Camouflagable components in the ISCAS-85 benchmarks when applying different camouflaging techniques. Note that, in the case of transformable interconnects, any net can be chosen, but the choice of dummy connections is restricted to avoid creating apparent combinational loops.
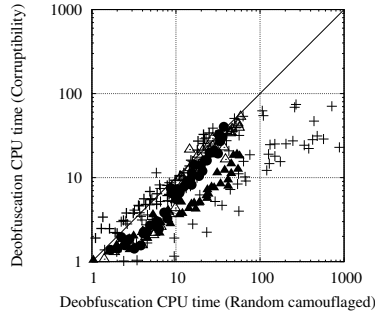


(a) Runtime of deobfuscation.
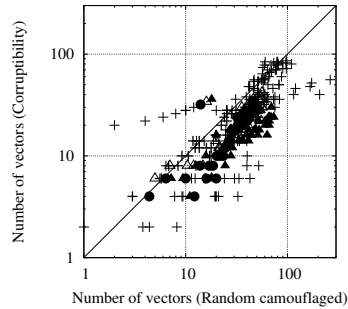
(b) Number of vectors of deobfuscation.

Figure 6.3: Comparing the total deobfuscation CPU time and the number of vectors used for deobfuscation of *Corruptibility*-guided and *random* camouflaged ISCAS-85 benchmarks.

## 6.2   Limitation of SAT-based De-obfuscation

SAT has the limited performance to solve certain circuit, such as multipliers [7]. The effectiveness of SAT-based deobfuscation on such circuits is also limited. To investigate this, I tested my algorithm on two multiplier. First, I use ten Camouflaged Standard Cells to obfuscate c6288, and it takes 5.4 hours to get solved. Most of the time is spent on the last iteration, which is the one SAT solver returns an UNSAT result (at line 6 of Alg. 1). I also repeat the same experiments on a 16-bit Montgomery multiplier [16] in field $GF(2^{16})$, and the program cannot finish solving in 6 hours with only one gate camouflaged. The reason is that solving GF multiplier problems using SAT is tough, which has been studied in [22]. So, including my work, any SAT-based algorithm has limited ability to reverse engineer multipliers or cryptographic ciphers.

This situation can be utilized by works that camouflaged circuits in a way that resists SAT-based reverse engineering [38].
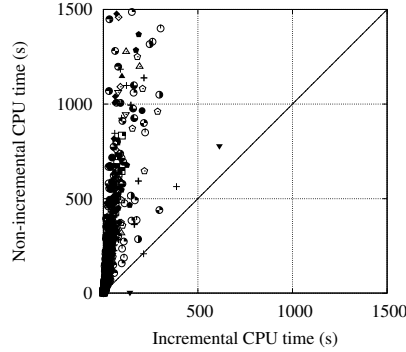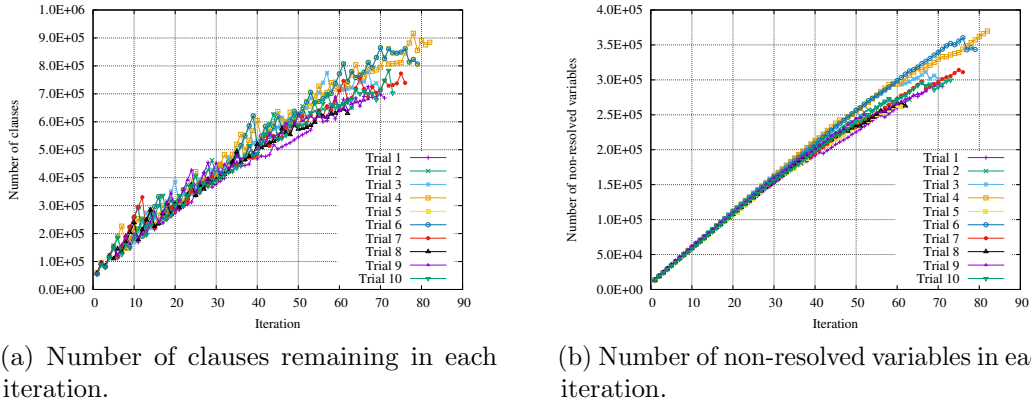


Figure 6.4: Comparing the total de-obfuscation runtime of baseline and incremental algorithms on 2400 randomly camouflaged circuits instances using different styles of camouflaged gates. The incremental solver gives an average speedup of 10.5x. Runtimes exceeding 1500 seconds are truncated from the plot.



(a) Number of clauses remaining in each iteration.



(b) Number of non-resolved variables in each iteration.

Figure 6.5: Examining the variable and clauses elimination using incremental SAT solving on 10 randomly camouflaged instances of ISCAS-85 benchmark c7552, each with 200 NAND/NOR/XOR camouflaged standard cells [27].

## 6.3 Incremental Algorithm versus Baseline

In this section, I provide the comparison between my work and the result from baseline, which is the algorithm from El Massad et al. [24]. Both of the programs can solve any kind of camouflaging, but my comparison focuses on *Camouflaged Standard*

*Cells* because it is used in El Massad's work. Fig. 6.4 is the runtime by using baseline and incremental algorithms to solve ISCAS-85 benchmarks with randomly selected gates that use XOR/NOR/NAND and the cells from Fig. 3.1b. Each dot stands for the runtime for my solver and the baseline. My solver can solve the largest example faster than baseline, where the incremental solver gives more significant improvement.

To quantify the efficiency benefit from the incremental program, I here provide more detail using the example that c7552 camouflaged with 200 Camouflaged Standard Cells [27]. Fig. 6.5 shows the trend of the number of clauses and the number of unsolved variables. Theoretically, a non-incremental solver doesn't make any further simplification. Thus both of the trends would appear linear as new copies of the CNF-encoded circuit are added to the original problem in each iteration. Subfig. 6.5a shows sub-linear growth in the number of clauses, and in Subfig. 6.5b, the number of unsolved variables also grows slower than linear. This figure suggests that some unnecessary variables and values are simplified.

## 6.4 Evaluating for combination of different camouflaging techniques

The previous section demonstrated the performance of each technique. Camouflaging a chip using Camouflaged Standard Cells [5] can provide functionality obfuscation, using Transformable Interconnection [4] can provide connection obfuscation, and using Obfusgates [23] can provide both functionality and connection obfuscation. However, the overall performance about combining different technique might be different. To investigate in which ratio combining the both functionality and connection obfuscation can provide the best security, I applied *NOR/NAND/XOR Camouflaged Standard Cell* (Fig.3.1(a)) and *Transformable Interconnect* (Fig. 3.3) to ISCAS-85 benchmarks as described here:

1. First, I apply *NOR/NAND/XOR Camouflaged Standard Cell* (Fig.3.1(a)) to replace the original gate to camouflage a circuit. I select gates to obfuscate by randomly.

2. Then I apply *Transformable Interconnect* (Fig.3.3). Similar to the previous section, I randomly choose nets to camouflage and then attach three dummy wires from the previous level to the 4-to-1 multiplexer. The four nets can be from the original circuits or the nets from Camouflaged Standard Cells.

3. Each Camouflage Standard Cell can provide three possibilities of the circuit, and each Transformable Interconnect can provide four possibilities to the circuit.

4. Except for $c499$, the number of camouflaged gates are 200. $C499$ does not have 200 gates, so I use 100 instead.

The result is shown in Fig. 6.6. Different from my expectation, there is not such a best ratio. Each benchmark favors to one single type of obfuscation rather than a combination. However, different circuit tends to various techniques. According to the result, obfuscation via functionality and connection may have similar effects on $c1355$. Functionality obfuscation appears to provide better security for $c499$ and $c880$, which are relatively smaller benchmarks. Moreover, connection obfuscation appears to work better in the other larger designs.
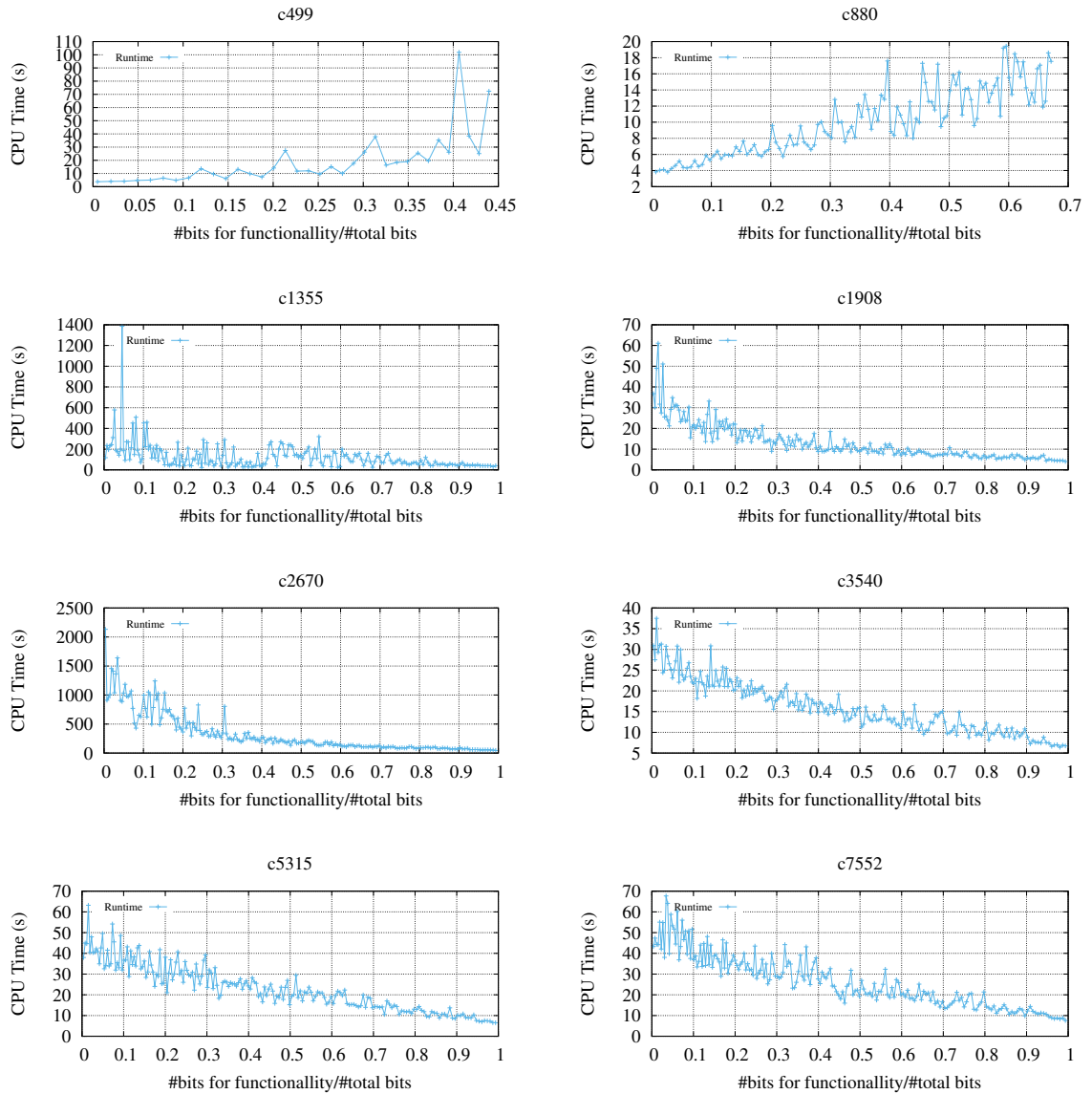
Figure 6.6: Plots show each design favors to a certain type of obfuscation. Smaller circuit tends to functionality obfuscation, while connection obfuscation works better with larger circuit.

# CHAPTER 7

# CONCLUSION

This work proposes an incremental-SAT based approach for de-obfuscating camouflaged circuits. I have implemented the algorithm and tested its performance by using it to de-obfuscate ISCAS-85 combinational benchmarks when camouflaged using three different styles of component camouflaging. The results show that the algorithm is able to efficiently deobfuscate the ISCAS-85 benchmarks regardless of camouflaging style, and is able to do so 10.5x faster than the best existing approaches. Our tool is released publicly to evaluate and support development of future selective component camouflaging approaches. [21] [40] [39]

# BIBLIOGRAPHY

[1] Becker, G. T., Regazzoni, F., Paar, C., and Burleson, W. P. Stealthy dopant-level hardware trojans. In *Cryptographic Hardware and Embedded Systems-CHES 2013*. Springer, 2013, pp. 197–214.

[2] Bi, Y., Shamsi, K., Yuan, J., Gaillardon, P., Micheli, G., Yin, X., Hu, X. S., Niemier, M., and Jin, Y. Emerging technology-based design of primitives for hardware security. *ACM Journal on Emerging Technologies in Computing Systems (JETC) 13*, 1 (2016), 3.

[3] Chakraborty, R.S., and Bhunia, S. Harpoon: an obfuscation-based soc design methodology for hardware protection. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 28*, 10 (2009), 1493–1502.

[4] Chen, S., Chen, J., Forte, D., Di, J., Tehranipoor, M., and Wang, L. Chip-level anti-reverse engineering using transformable interconnects. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS), 2015 IEEE International Symposium on* (2015), IEEE, pp. 109–114.

[5] Chow, L., Baukus, J.P., and Jr, W. M. Clark. Integrated circuits protected against reverse engineering and method for fabricating the same using an apparent metal contact line terminating on field oxide, Nov. 13 2007. US Patent 7,294,935.

[6] Cocchi, R. P., Baukus, J. P., Chow, L. W., and Wang, B. J. Circuit camouflage integration for hardware IP protection. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE* (2014), IEEE, pp. 1–5.

[7] Cook, S. A., and Mitchell, D. G. Finding hard instances of the satisfiability problem. In *Satisfiability Problem: Theory and Applications: DIMACS Workshop, March 11-13, 1996* (1997), vol. 35, American Mathematical Soc., p. 1.

[8] Eén, N., and Sörensson, N. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science 89*, 4 (2003), 543–560.

[9] Een, N., and Sörensson, N. An extensible SAT-solver. *Theory and Applications of Satisfiability Testing* (2004).

[10] Gascón, A., Subramanyan, P., Dutertre, B., Tiwari, A., Jovanovic, D., and Malik, S. Template-based circuit understanding. In *Formal Methods in Computer-Aided Design (FMCAD), 2014* (2014), IEEE, pp. 83–90.

[11] Hansen, M.C., Yalcin, H., and Hayes, J.P. Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering. *Design Test of Computers, IEEE 16*, 3 (1999), 72–80.

[12] Iyengar, A., and Ghosh, S. Threshold voltage-defined switches for programmable gates.

[13] Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. Oracle-guided component-based program synthesis. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on* (2010), vol. 1, IEEE, pp. 215–224.

[14] Keshavarz, S., and Holcomb, D. Privacy leakages in approximate adders. In *International Symposium of Circuits and Systems* (2017).

[15] Keshavarz, S., Paar, C., and Holcomb, D. Design automation for obfuscated circuits with multiple viable functions. In *Design Automation and Test in Europe (DATE'17)* (2017).

[16] Koc, C.K., and Acar, T. Montgomery multiplication in gf (2k). *Designs, Codes and Cryptography 14*, 1 (1998), 57–69.

[17] Kömmerling, O., and Kuhn, M. G. Design principles for tamper-resistant smart-card processors. *Smartcard 99* (1999), 9–20.

[18] Larrabee, T. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 11*, 1 (1992), 4–15.

[19] Li, M., K.Shamsi, Meade, T., Zhao, Z., Yu, B., Jin, Y., and Pan, D. Z. Provably secure camouflaging strategy for IC protection. In *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016* (2016), p. 28.

[20] Li, W., Wasson, Z., and Seshia, S. A. Reverse engineering circuits using behavioral pattern mining. In *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on* (2012), pp. 83–88.

[21] Liu, D., Zhang, X., Yu, C., and Holcomb, D. Oracle-Guided Incremental SAT Solving to Reverse Engineer Camouflaged Logic Circuits. *IEEE/ACM/EDAA Design, Automation and Test in Europe (DATE)* (2016).

[22] Lv, J., Kalla, P., and Enescu, F. Efficient gröbner basis reductions for formal verification of galois field multipliers. In *Proceedings of the Conference on Design, Automation and Test in Europe* (2012), EDA Consortium, pp. 899–904.

[23] Malik, S., Becker, G., Paar, C., and Burleson, W. Development of a Layout-Level Hardware Obfuscation Tool. In *IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2015* (2015).

[24] Massad, M. E., Garg, S., and Tripunitara, M. V. Integrated circuit (IC) decamouflaging: Reverse engineering camouflaged ics within minutes. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014* (2015).

[25] Rahmati, A., Hicks, M., Holcomb, D. E., and Fu, K. Probable cause: the deanonymizing effects of approximate dram. In *ACM SIGARCH Computer Architecture News* (2015), vol. 43, ACM, pp. 604–615.

[26] Rajendran, J., Pino, Y., Sinanoglu, O., and Karri, R. Logic encryption: A fault analysis perspective. In *Proceedings of the Conference on Design, Automation and Test in Europe* (2012), DATE '12, pp. 953–958.

[27] Rajendran, J., Sam, M., Sinanoglu, O., and Karri, R. Security analysis of integrated circuit camouflaging. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 709–720.

[28] Shiozaki, M., Hori, R., and Fujino, T. Diffusion programmable device : The device to prevent reverse engineering. *IACR Cryptology ePrint Archive 2014* (2014), 109.

[29] Subramanyan, P., Ray, S., and Malik, S. Evaluating the security of logic encryption algorithms. In *Hardware-Oriented Security and Trust (HOST)* (2015).

[30] Subramanyan, P., Tsiskaridze, N., Pasricha, K., Reisman, D., Susnea, A., and Malik, S. Reverse engineering digital circuits using functional analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe* (2013), EDA Consortium, pp. 1277–1280.

[31] SypherMedia. Syphermedia library – circuit camouflage technology. `http://www.smi.tv/SMI_SypherMedia_Library_Intro.pdf`, 2012. Online; accessed 21-April-2015.

[32] Torrance, R., and James, D. The state-of-the-art in semiconductor reverse engineering. In *Proceedings of the 48th Design Automation Conference* (2011), DAC '11, pp. 333–338.

[33] Vijayakumar, A., Patil, V., Holcomb, D. E., Paar, C., and Kundu, S. Physical design obfuscation of hardware: A comprehensive investigation of device and logic-level techniques. *IEEE Transactions on Information Forensics and Security 12*, 1 (2017), 64–77.

[34] Xie, Y., and Srivastava, A. Mitigating sat attack on logic locking. In *Cryptographic Hardware and Embedded Systems-CHES 2016*. Springer, 2016.

[35] Yang, B., Wu, K., and Karri, R. Scan based side channel attack on dedicated hardware implementations of data encryption standard. In *Test Conference, 2004. Proceedings. ITC 2004. International* (2004), IEEE, pp. 339–344.

[36] Yasin, M., Mazumdar, B., Rajendran, J., and Sinanoglu, O. Sarlock: Sat attack resistant logic locking. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (2016), IEEE, pp. 236–241.

[37] Yasin, M., Mazumdar, B., Sinanoglu, O., and Rajendran, J. Camoperturb: secure IC camouflaging for minterm protection. In *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016* (2016), p. 29.

[38] Yasin, M., Rajendran, J., Sinanoglu, O., and Karri, R. On improving the security of logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems PP*, 99 (2015), 1–1.

[39] Yu, C., Holcomb, D., and Ciesielski, M. Reverse Engineering Irreducible Polynomial of GF($2^m$) Arithmetic. *IEEE/ACM/EDAA Design, Automation and Test in Europe (DATE)* (2017).

[40] Yu, C., Zhang, X., Liu, D., Ciesielski, M., and Holcomb, D. Incremental SAT-based Reverse Engineering of Camoulaged Logic Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017).