Functional Analysis Attacks on Logic Locking

Deepak Sirone Pramod Subramanyan
Indian Institute of Technology, Kanpur
{*dsirone, spramod*}@cse.iitk.ac.in

Abstract

This paper proposes Functional Analysis attacks on state of the art Logic Locking algorithms (abbreviated as Fall attacks). Fall attacks have two stages. The first stage identifies nodes involved in the locking functionality and analyzes functional properties of these nodes to shortlist a small number of candidate locking keys. In many cases, this shortlists exactly one locking key, so no further analysis is needed. However, if more than one key is shortlisted, the second stage introduces a SAT-based algorithm to identify the correct locking key from a list of alternatives using simulations on an unlocked circuit.

In comparison to past work, the Fall attack is more practical as it can often succeed (90% of successful attempts in our experiments) by only analyzing the locked netlist, without requiring oracle access to an unlocked circuit. Further, Fall attacks successfully defeat Secure Function Logic Locking (SFLL), the only locking algorithm that is resilient to known attacks on logic locking. Our experimental evaluation shows that Fall is able to defeat 65 out of 80 (81%) circuits locked using SFLL.

1 Introduction

Globalization and concomitant de-verticalization of the semiconductor supply chain have resulted in IC design houses becoming increasingly reliant on potentially untrustworthy offshore foundries. This reliance has raised concerns of integrated circuit (IC) piracy, unauthorized overproduction, and malicious design modifications by adversarial entities that may be part of these contract foundries [7, 10, 19]. These issues have both financial [8] and national security implications [16].

A potential solution to these problems is logic locking [2, 13]: a set of techniques that introduce additional logic and new inputs to a digital circuit in order to create a "locked" version of it. The locked circuit operates correctly if and only if the new inputs, referred to as "key inputs" are set to the right values. Typically, key inputs are connected to a tamper-proof memory. The circuit is activated by the design house by programming the correct key values after manufacturing and prior to sale. The security assumption underlying logic locking

is that the *adversary (untrusted foundry) does not know the correct values of the key inputs and cannot compute it.*

Initial proposals for logic locking did not satisfy this assumption and were vulnerable to attack [11, 12, 17, 23, 26]. For example, Rajendran et al. [12] used automatic test pattern generation (ATPG) tools to compute input values that would allow an adversary to reveal the values of key bits. Subramanyan et al. [17] developed the SAT attack which defeated all known logic encryption techniques at the time. The SAT attack works by using a Boolean SATisfiability solver to iteratively find inputs that distinguish between equivalence classes of keys. For each such input, an activated IC (perhaps purchased from the market by the adversary) is queried for the correct output and this information is fed back to the SAT solver when computing the next distinguishing input. The practicality of this attack depends on the number of equivalence classes of keys present in the locked circuit.

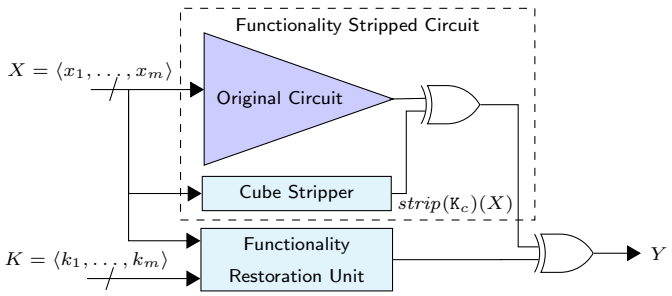

Figure 1: Overview of SAT attack resilient locking algorithms like TTLock and SFLL-HD^h. We show a single output circuit for simplicity, additional outputs are handled symmetrically.

Much subsequent work has focused on SAT attack resilient logic locking [20, 21, 24, 27, 28]. These proposals attempt to guarantee that the number of equivalence classes of keys is exponential in the key length. Broadly speaking, they have the structure shown in Figure 1. They introduce a circuit which "flips" the output of the original circuit for a particular cube or set of cubes. We refer to this component as the *cube stripping unit.* This flipped output is then inverted by a key-dependent circuit that we refer to as the *progammable functionality restoration unit.* This latter circuit is guaranteed to have an exponential number of equivalence classes of keys and ensures SAT attack resilience.[1] Initial proposals along these lines were Anti-SAT [20, 21] and SARLock [24]. However, Anti-SAT was vulnerable to the signal probability skew (SPS) [24] attack while SARLock was vulnerable to the Double DIP [15] attack and the Approximate SAT [14] attack. Both schemes are vulnerable to removal and bypass attacks [22, 25]. Subsequently, Yasin et al. proposed TTLock [28] and Secure

[1]For these schemes to work as intended, the locking key has to be "hard coded" in the cube stripping unit. We exploit this vulnerability.

Function Logic Locking (SFLL) [27]. To the best of our knowledge, SFLL is the only logic locking technique that is resilient to all of the above attacks.

1.1 Contributions

In this paper, we introduce a novel class of Functional Analysis attacks on Logic Locking (abbreviated as FALL attacks). FALL attacks defeat locking methods which use cube stripping and programmable functionality restoration.

Our approach is to use **structural and functional analyses** of circuit nodes to first identify the gates that are the output of the cube stripping module. There are two challenges involved in this. First, the locked netlist is a sea of gates, and examining every gate using computationally expensive functional analyses is not feasible. Second, testing for whether a gate is equivalent to the cube stripping function for some key value involves solving a quantified Boolean formula (QBF). QBF is PSPACE-complete [1] in comparison to to SAT which is "only" NP-complete [5], and therefore the naïve approach does not even scale to small netlists. Our first contribution tackles these problems by the development of a set of functional properties of the cube stripping function used in SFLL. We then use SAT-based analyses to find nodes with these properties and determine a shortlist of potential locking keys. These functional analyses are able to defeat SFLL, which to the best of our knowledge, is the only locking method resilient to all known attacks.

In about 90% of successful attempts in our experiments, the first stage of the attack shortlists exactly one potential key. In such cases, the FALL attack **does not require input/output (I/O) oracle access** to an unlocked circuit. Any malicious foundry who can reconstruct gate-level structures from the masks can use FALL without setting up logic analyzers, loading the scan chain, etc. Our second contribution is evidence supporting the claim that attacking logic locking may be much easier than previously believed.

Our third contribution is a novel SAT-based **key confirmation algorithm**. Given a list of suspected key values and I/O oracle access, key confirmation can be used to prove that one (or none) of these suspected key values is correct. This has two important implications. Key confirmation can be used in isolation and provides a powerful new tool in the hands of attackers: attacker need only guess a key value through an arbitrary circuit analysis and key confirmation can be used to verify this guess. Second, the key confirmation algorithm succeeds on netlists that are resilient to the SAT attack, thus providing a new path toward the use of powerful Boolean reasoning engines in the security analysis of logic locking.

Our final contribution is a thorough experimental analysis of the FALL attack which shows that our attacks succeed on 65 out of 80 benchmark circuits (81%) in our evaluation. Among these 65, the functional analysis shortlists exactly one key for 58 circuits (90% of successful attempts), supporting our claim that Oracle-less attacks are indeed practical. Finally, we show experimentally that our key confirmation attack succeeds on all the benchmark circuits we examine and is orders of magnitude faster than the SAT attack [17].

2 Attack Overview

This section first describes the adversary model for the FALL attack. It then provides an overview of the attack itself and describes the notation used in the rest of this paper.

2.1 Adversary Model

The adversary is assumed to be a malicious foundry with layout and mask information. The gate level netlist can be reverse engineered from this [18]. The adversary knows the locking algorithm and its parameters (e.g., h in SFLL-HD^h) and can distinguish between key inputs and circuit inputs. We assume that the adversary may have access to an activated circuit which can be used to observe the output for a specific input. We follow [12, 17, 27] etc. and restrict our attention to combinational circuits. Sequential circuits can viewed as combinational by treating flip-flop inputs and outputs as combinational outputs and inputs respectively.

2.2 Overview of Attack Stages

Figure 2 shows the three main stages of the FALL attack.

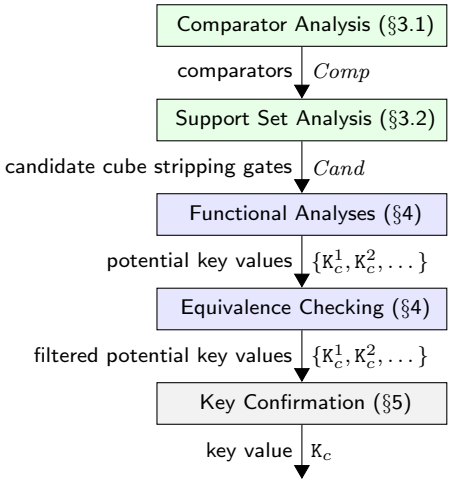

Figure 2: Attack algorithm overview.

The first stage uses largely structural analyses to identify candidate gates that may be the output of a cube stripping module. This is described in Section 3

The second stage subjects these candidate nodes to functional analysis to identify suspected key values. Algorithms for functional analysis exploit unateness and Hamming distance properties of the cube stripping functions used in SFLL and are described in Section 4.

Given a shortlist of suspected key values, the third stage verifies whether one of these key values is correct using the key confirmation algorithm described in Section 5. This stage of the attack need not be carried out if only one key was identified by the functional analyses or if the adversary does not have I/O oracle access to an activated circuit.

2.3 Notation

$\mathbb{B} = \{0, 1\}$ is the Boolean domain. A combinational logic circuit is modeled as a directed acyclic graph $G = (V, E)$. Nodes in the graph correspond to logic gates and input nodes. Edge $(v_1, v_2) \in E$ if v_2 is a fanin (input) of the gate v_1.

Given a node $v \in V$, define $\mathit{fanins}(v) = \{v' \mid (v, v') \in E\}$. $\#\mathit{fanins}(v)$ is the cardinality of $\mathit{fanins}(v)$. For $v \in V$ such that $\#\mathit{fanins}(v) = n$, $\mathit{nodefn}(v)$ is the n-ary Boolean function associated with the node; $\mathit{nodefn}(v) : V \to (\mathbb{B}^n \to \mathbb{B})$. For example, if v_1 is a 2-input AND gate, $\mathit{nodefn}(v_1) = \lambda ab.\ a \wedge b$. For input nodes, $\mathit{nodefn}(v)$ is an uninterpreted 0-ary Boolean function (or equivalently, a propositional variable). The circuit function of node v, denoted $\mathit{cktfn}(v)$ is defined recursively as: $\mathit{cktfn}(v) = \mathit{nodefn}(v)(\mathit{cktfn}(v_1), \ldots, \mathit{cktfn}(v_n))$ where $v_i \in \mathit{fanins}(v)$. The transitive fanin cone of a node v, denoted $\textsc{Tfc}(v)$, is the set of all nodes v_j such that $(v, v_j) \in E$ or there exists some $v_i \in V$ such that $(v_i, v_j) \in E$ and $v_i \in \textsc{Tfc}(v)$. The support of a node, denoted by $\mathit{Supp}(v)$, is the set of all nodes v_j such that $v_j \in \textsc{Tfc}(v)$ and $\#\mathit{fanins}(v_j) = 0$.

In a locked netlist, some input nodes are specially distinguished key inputs. Define the predicate $\mathit{isKey}(v)$ such that $\mathit{isKey}(v) = 1$ iff and node $v \in V$ is a key input.

Given two bit vectors $X^1 = \langle x_1^1, \ldots, x_m^1 \rangle$ and $X^2 = \langle x_1^2, \ldots, x_m^2 \rangle$, define $HD(X^1, X^2) \doteq \sum_{i=1}^{m} (x_i^1 \oplus x_i^2)$ to be their Hamming distance. $\oplus$ is the eXclusive OR operator, and $\sum$ is bit vector sum.

Finally, given a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$, the function obtained by setting $x_i = 1$ in f, $f(x_1, \ldots, 1, \ldots, x_n)$, denoted as f_{x_i} is called a positive cofactor of f. $f(x_1, \ldots, 0, \ldots, x_n)$ denoted $f_{\neg x_i}$, is a negative cofactor of f.

3 Structural Analyses

This section describes structural analyses to identify nodes that may be the output of the cube stripping unit.

3.1 Comparator Identification

Comparator identification finds all nodes in the circuit which are the result of comparing an input value with a key input. Such nodes are very likely to be part of the functionality restoration unit. Stated precisely, we wish to identify all gates whose circuit function is equivalent to $(z \oplus x_i) \iff k_i$ for some z. Here x_i must be a circuit input, k_i must be a key input and z captures whether k_i is being compared with x_i or $\neg x_i$.

The result of comparator identification is the set $Comp = \{\langle v_i, x_i, k_i\rangle, \dots\}$ where each tuple $\langle v_i, x_i, k_i\rangle$ is such that $Supp(v_i) = \{x_i, k_i\}$, $isKey(x_i) = 0$, $isKey(k_i) = 1$, and one of the following two formulas is valid: (i) $cktfn(v_i) \iff x_i \oplus k_i$ and (ii) $cktfn(v_i) \iff \neg(x_i \oplus k_i)$.

3.2 Support Set Matching

The set of all input nodes x_i that appear in *Comp* should be the support of the cube stripping unit. Support set matching finds all such nodes. Given the set $Comp = \{\langle v_i, x_i, k_i, \rangle, \dots\}$, define the projection $Comp_x$ as $Comp_x = \{x_i \mid (v_i, x_i, k_i) \in Comp\}$. The set *Cand* is set of all gates whose support is identical to $Comp_x$. This set of gates must contain the output of the cube stripping unit.

4 Functional Analyses

This section first develops functional properties of the cube stripping function used in SFLL. It then describes three algorithms that exploit these properties to find the "hidden" key input parameters of the cube stripping unit.

4.1 Functional Properties of Cube Stripping

Cube stripping involves the choice of a protected cube, represented by the tuple $\mathtt{K}_c = \langle \mathtt{k}_1, \dots, \mathtt{k}_m\rangle$ where $m = |Comp|$ and $\mathtt{k}_i \in \mathbb{B}$. A stripping function $strip : \mathbb{B}^m \to (\mathbb{B}^m \to \mathbb{B})$ is parameterized by this protected cube. The output of the functionality stripped circuit (the dashed box in Figure 1) is inverted for the input $X = \langle x_1, \dots, x_m\rangle$ when $strip(\mathtt{K}_c)(X) = 1$. For a given locked circuit and associated key value, the value of $\mathtt{K}_c$ is "hard-coded" into the implementation of *strip*, which is why we typeset $\mathtt{K}_c$ in a fixed width font. The attacker's goal is to learn this value of $\mathtt{K}_c$.

In this paper we study functional properties of the following cube stripping function: $strip_h(\mathtt{K}_c)(X) \doteq HD(\mathtt{K}_c, X) = h$. $strip_h$ flips the output for all input patterns exactly Hamming distance h from the protected cube $\langle \mathtt{k}_1, \dots, \mathtt{k}_m\rangle$. This is the cube stripping function for SFLL-HD^h and the special case of $h = 0$ corresponds to the cube stripping function for TTLock. This function has three specific properties that can be exploited to determine the value of $\mathtt{K}_c$.

4.1.1 Unateness (TTLock/SFLL-HD^0)

We say that a Boolean function $f(x_1, \dots, x_m) : \mathbb{B}^m \to \mathbb{B}$ is positive unate in the variable x_i if $f(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots) \leq f(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots)$. We say that f is negative unate in the variable x_i if $f(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots) \leq f(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots)$. Function f is said to be unate in x_i if it is either positive or negative unate in x_i.[2]

[2] $a \leq b$ is defined as $\neg a \vee b$.

(Lemma 1) The cube stripping function for TTLock/SFLL-HD^0 is unate in every variable x_i. Further, it is positive unate in x_i if $\mathtt{k}_i = 1$ and negative unate in x_i if $\mathtt{k}_i = 0$.

For example, let $\langle \mathtt{k}_1, \mathtt{k}_2, \mathtt{k}_3 \rangle = \langle 1, 0, 1 \rangle$. Then $\mathit{strip}_0(\mathtt{k}_1, \mathtt{k}_2, \mathtt{k}_3)(x_1, x_2, x_3) = x_1 \wedge \neg x_2 \wedge x_3$. This is positive unate in x_1 as $0 \leq \neg x_2 \wedge x_3$, and negative unate in x_2 as $0 \leq x_1 \wedge x_3$.

4.1.2 Non-Overlapping Errors Property (SFLL-HD^h)

Consider the definition of strip_h, let $\mathtt{K}_c = \langle \mathtt{k}_1, \ldots, \mathtt{k}_4 \rangle = \langle 1, 1, 1, 1 \rangle$ and $h = 1$. Consider the two input values $\mathtt{X}^1 = \langle 1, 1, 1, 0 \rangle$ and $\mathtt{X}^2 = \langle 0, 1, 1, 1 \rangle$. $\mathit{strip}_1(\mathtt{K}_c)(\mathtt{X}^1) = 1 = \mathit{strip}_1(\mathtt{K}_c)(\mathtt{X}^2)$. $\mathtt{X}^1$ and $\mathtt{X}^2$ are Hamming distance 2 apart. Due to the definition of strip_1 they are also Hamming distance 1 from $\mathtt{K}_c$. This means that the values of x_i on which the two patterns agree – x_2 and x_3 – must be equal to $\mathtt{k}_2$ and $\mathtt{k}_3$ respectively. This is because the "errors" in $\mathtt{X}^1$ and $\mathtt{X}^2$ cannot overlap as they are Hamming distance $2h$ apart. Generalizing this observation leads to the following result.

(Lemma 2) Suppose $\mathtt{X}^1 = \langle \mathtt{x}^1_1, \ldots, \mathtt{x}^1_m \rangle$, $\mathtt{X}^2 = \langle \mathtt{x}^2_1, \ldots, \mathtt{x}^2_m \rangle$, $\mathtt{K}_c = \langle \mathtt{k}_1, \ldots, \mathtt{k}_m \rangle$ and $\mathit{strip}_h(\mathtt{K}_c)(\mathtt{X}^1) = 1 = \mathit{strip}_h(\mathtt{K}_c)(\mathtt{X}^2)$. If $\mathit{HD}(\mathtt{X}^1, \mathtt{X}^2) = 2h$, then for every j such that $\mathtt{x}^1_j = \mathtt{x}^2_j$, we must have $\mathtt{x}^1_j = \mathtt{x}^2_j = \mathtt{k}_j$.

4.1.3 Sliding Window Property (SFLL-HD^h)

Let us revisit the example from the non-overlapping errors property. Let $\mathtt{K}_c = \langle \mathtt{k}_1, \ldots, \mathtt{k}_4 \rangle = \langle 1, 1, 1, 1 \rangle$ and $h = 1$. For the input value $\mathtt{X}^1 = \langle 1, 1, 1, 0 \rangle$, we have $\mathit{strip}_1(\mathtt{K}_c)(\mathtt{X}^1) = 1$. Notice that *there cannot exist* another assignment $\mathtt{X}^2 = \langle \mathtt{x}^2_1, \ldots, \mathtt{x}^2_4 \rangle$ with $\mathtt{x}^2_4 = 0$, $\mathit{HD}(\mathtt{X}^1, \mathtt{X}^2) = 2$ and $\mathit{strip}_1(\mathtt{K}_c)(\mathtt{X}^2) = 1$. This is because $\mathtt{x}^2_4 \neq \mathtt{k}_4$, so the remaining bits in $\mathtt{X}^2$ must be equal to $\mathtt{K}_c$ so that $\mathit{strip}_1(\mathtt{K}_c)(\mathtt{X}^2) = 1$. But this forces the Hamming distance between $\mathtt{X}^1$ and $\mathtt{X}^2$ to be 0 (and not 2 as desired). This observation leads to the following result.

(Lemma 3) Consider the assignments $\mathtt{X}^1 = \langle \mathtt{x}^1_1, \ldots, \mathtt{x}^1_m \rangle$ and $\mathtt{X}^2 = \langle \mathtt{x}^2_1, \ldots, \mathtt{x}^2_m \rangle$. Let $\mathtt{K}_c = \langle \mathtt{k}_1, \ldots, \mathtt{k}_m \rangle$ as before. The formula $\mathit{strip}_h(\mathtt{K}_c)(\mathtt{X}^1) = 1 \wedge \mathit{strip}_h(\mathtt{K}_c)(\mathtt{X}^2) = 1 \wedge \mathit{HD}(\mathtt{X}^1, \mathtt{X}^2) = 2h \wedge \mathtt{x}^1_j = \mathtt{x}^2_j \wedge \mathtt{x}^1_j = \mathtt{b}$ is satisfiable iff $\mathtt{b} = \mathtt{k}_j$.

4.2 Functional Analysis Algorithms

In this subsection, we describe three attack algorithms on SFLL that are based on Lemmas 1, 2 and 3. Each algorithm takes as input a candidate node c in the circuit DAG. Let $X = \mathit{Supp}(c)$. The functional analyses described in this subsection determine whether the circuit function of this node $\mathit{cktfn}(c)(X)$ is equivalent to $\mathit{strip}(\mathtt{K}_c)(X)$ for some assignment to $\mathtt{K}_c$. In other words, we are trying to solve the quantified Boolean formula (QBF): $\exists \mathtt{K}_c.\ \forall X.\ \mathit{cktfn}(c)(X) \iff \mathit{strip}(\mathtt{K}_c)(X)$. However, solving this QBF instance is computationally hard. So instead we exploit Lemmas 1, 2 and 3 to determine potential values of $\mathtt{K}_c$ and verify this "guess" using combinational equivalence checking.

4.2.1 AnalyzeUnateness

This is shown in Algorithm 1 and can be used to attack SFLL-HD^0.

Algorithm 1 Algorithm AnalyzeUnateness

```
1: procedure AnalyzeUnateness(c)
2:     keys ← ∅
3:     for x_i ∈ Supp(c) do
4:         if isPositiveUnate(c, x_i) then
5:             keys ← keys ∪ (x_i ↦ 1)
6:         else if isNegativeUnate(c, x_i) then
7:             keys ← keys ∪ (x_i ↦ 0)
8:         else return ⊥
9:         end if
10:    end for
11:    return keys
12: end procedure
```

It takes as input a circuit node c and outputs an assignment to each node in the support set of c if the function represented by c is unate, otherwise it returns $\perp$. This assignment is the protected cube in TTLock/SFLL-HD^0.

4.2.2 SlidingWindow

This is shown in Algorithm 2 and can be used to attack SFLL-HD^h for $h < \lfloor m/2 \rfloor$; m is the number of key inputs.

Again, the input is circuit node c and the algorithm checks if c behaves as a Hamming distance calculator in the cube stripping unit of SFLL-HD^h. It works by asking if there are two distinct satisfying assignments to $cktfn(c)$ which are Hamming distance of $2h$ apart. If no such assignment exists then $\perp$ is returned. Otherwise, by Lemma 2, bits which are equal in both satisfying assignments must also be equal to the corresponding key bits. The remaining bits are obtained by iterating through each remaining bit and applying the SAT query in Lemma 3. If any query is inconsistent with Lemma 3 during this process then $\perp$ is returned. If successful, the return value is again the protected cube.

4.2.3 Distance2H

This is shown in Algorithm 3. It is based on Lemma 2 and is applicable when $4h \leq m$; m is the number of key inputs.

This procedure is similar to SlidingWindow in that it computes two satisfying assignments to c that are distance of $2h$ apart. Any bits that are equal between the two assignments must be equal to the key bits. The remaining bits are computed by asking if there are two more satisfying assignments such that the bits which were not equal in the first pair of assignments are now equal.

Algorithm 2 Algorithm SLIDINGWINDOW

procedure SLIDINGWINDOW(c)
  $keys \leftarrow \emptyset$
  $S \leftarrow Supp(c)$
  $c' \leftarrow subsitute(c, \{(x_i, x'_i) \mid x \in S\})$
  $F \leftarrow c \wedge c' \wedge HD(Supp(c), Supp(c')) = 2h$
  if $solve(F) = \texttt{UNSAT}$ **then return** $\perp$
  end if
  for $x_i \in S$ **do**
    $(\texttt{m}_i, \texttt{m}'_i) \leftarrow (\texttt{model}_{x_i}(F), \texttt{model}_{x'_i}(F))$
    if $\texttt{m}_i = \texttt{m}'_i$ **then**
      $keys \leftarrow keys \cup (x_i \mapsto \texttt{m}_i)$
    else
      $r_i \leftarrow solve(F \wedge (x_i = x'_i \wedge x'_i = \texttt{m}_i))$
      $r'_i \leftarrow solve(F \wedge (x_i = x'_i \wedge x'_i = \texttt{m}'_i))$
      if $r_i = \texttt{SAT} \wedge r'_i = \texttt{UNSAT}$ **then**
        $keys \leftarrow keys \cup (x_i \mapsto \texttt{m}_i)$
      else if $r_i = \texttt{UNSAT} \wedge r'_i = \texttt{SAT}$ **then**
        $keys \leftarrow keys \cup (x_i \mapsto \texttt{m}'_i)$
      else
        return $\perp$
      end if
    end if
  end for
  return $keys$
end procedure

These new assignments must also be Hamming distance of $2h$ apart. The second query, if successful, determines the remaining key bits by Lemma 3. Note that DISTANCE2H is not applicable when $4h > m$, where h is the parameter in SFLL-HD^h and m is the number of key inputs.

5 Key Confirmation

The key confirmation algorithm takes as input a circuit described by the characteristic function of its input/output relation C, a predicate over the key values $\varphi : K \to \mathbb{B}$, and an I/O oracle. The algorithm either returns a key value $\texttt{K}_c$ s.t. $\texttt{K}_c \models \varphi$ or $\perp$ if no key value is consistent with φ and the oracle. The predicate φ is a Boolean formula over the key variables that constrains the search space of the algorithm. For example, suppose the circuit analyses have shortlisted two keys $\langle 1, 1, 0, 1\rangle$ and $\langle 0, 0, 1, 0\rangle$. The $\varphi(K) \doteq (k_1 = 1 \wedge k_2 = 1 \wedge k_3 = 0 \wedge k_4 = 1) \vee (k_1 = 0 \wedge k_2 = 0 \wedge k_3 = 1 \wedge k_4 = 0)$.

Algorithm 3 Algorithm DISTANCE2H

1: **procedure** DISTANCE2H(c)
2: $S \leftarrow Supp(c)$
3: $c' \leftarrow subsitute(c, \{(x_i, x_i') \mid x \in S\})$
4: $F \leftarrow c \wedge c' \wedge HD(Supp(c), Supp(c')) = 2h$
5: **if** $solve(F) = \texttt{UNSAT}$ **then return** $\perp$
6: **end if**
7: $M_F \leftarrow \{(x_i, \texttt{model}_{x_i}(F), \texttt{model}_{x_i'}(F)) \mid x_i \in S\}$
8: $keys_A \leftarrow \{(x_i \mapsto \texttt{m}_i) \mid (x_i, \texttt{m}_i, \texttt{m}_i') \in M_F \wedge \texttt{m}_i = \texttt{m}_i'\}$
9: $Cnst \leftarrow \{(x_i = x_i') \mid (x_i, \texttt{m}_i, \texttt{m}_i') \in M_F \wedge \texttt{m}_i \neq \texttt{m}_i'\}$
10: $G \leftarrow F \wedge (\bigwedge_{p_i \in Cnst} p_i)$
11: **if** $solve(G) = \texttt{UNSAT}$ **then return** $\perp$
12: **end if**
13: $M_G \leftarrow \{(x_i, \texttt{model}_{x_i}(G), \texttt{model}_{x_i'}(G)) \mid x_i \in S\}$
14: $keys_B \leftarrow \{(x_i \mapsto \texttt{m}_i) \mid (x_i, \texttt{m}_i, \texttt{m}_i') \in M_G \wedge \texttt{m}_i = \texttt{m}_i'\}$
15: **return** $keys_A \cup keys_B$
16: **end procedure**

5.1 Algorithm Description

Key confirmation is shown in Algorithm 4. The two main components of the algorithm are the sequences of formulas P_i and Q_i, which we implemented using two SAT solver objects. P_i are used to produce *candidate* key values that are consistent with φ. Note that since P_1 is φ, all subsequent $P_i \implies \varphi$. Q_i is used to generate distinguishing inputs. When P_i becomes UNSAT, it means no key value is consistent with φ and the oracle. Or equivalently, this means that the initial "guess" encoded in φ was incorrect. The algorithm terminates with a correct key when Q_i becomes UNSAT, i.e. no more distinguishing inputs can be found.

The two significant differences from the SAT attack [17] are: (i) the two solver objects corresponding to P_i and Q_i which helps separate the generation of candidate keys from the generation of distinguishing inputs, and (ii) the restriction that $P_i \implies \varphi$. The former allows us to differentiate between no key value being consistent with φ (line 6) from no distinguishing inputs being found (line 10) – this would not be possible in the SAT attack formulation. The latter ensures that instead of searching over the entire space of distinguishing inputs, we restrict the search to keys which satisfy φ.

5.2 Correctness of Key Confirmation

Correctness of key confirmation is captured by the following lemma.

(Lemma 4) Algorithm 4 terminates and returns either (i) the key $\texttt{K}_c$ or (ii) $\perp$. The former occurs iff $\texttt{K}_c \models \varphi$ and $\forall X.\ C(X, \texttt{K}_c, Y) \iff Y = oracle(X)$. The latter occurs iff no such $\texttt{K}_c$ exists.

The second clause of Lemma 4 is important to emphasize. Key confirmation

Algorithm 4 Key Confirmation Algorithm

procedure KeyConfirmation($C, \varphi, oracle$)
  $i \leftarrow 1$
  $P_1 \leftarrow \varphi$
  $Q_1 \leftarrow C(X, K_1, Y_2) \wedge C(X, K_2, Y_2) \wedge Y_1 \neq Y_2$
  while true **do**
    if $solve[P_i] = \texttt{UNSAT}$ **then**
      return $\perp$
    end if
    $\texttt{K}_i^1 \leftarrow \texttt{model}_{K_1}(P_i)$
    if $solve[Q_i \wedge (K_1 = \texttt{K}_i^1)] = \texttt{UNSAT}$ **then**
      return $\texttt{K}_i^1$
    end if
    $\texttt{X}_i^d \leftarrow \texttt{model}_X(Q_i)$
    $\texttt{Y}_i^d \leftarrow oracle(\texttt{X}_i^d)$
    $P_{i+1} \leftarrow P_i \wedge C(\texttt{X}_i^d, K_1, \texttt{Y}_i^d)$
    $Q_{i+1} \leftarrow Q_i \wedge C(\texttt{X}_i^d, K_2, \texttt{Y}_i^d)$
    $i \leftarrow i + 1$
  end while
end procedure

terminates with the result $\perp$ iff no key value $\texttt{K}_c$ s.t. $\texttt{K}_c \models \varphi$ is correct for the given oracle. This implies key confirmation can be safely used even if the key value was "incorrectly" guessed – the algorithm will detect this.

6 Evaluation

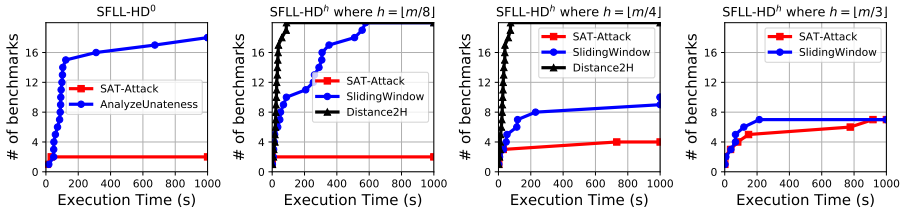

Figure 3: Circuit analyses: execution time vs number of benchmarks solved in that time.

This section describes our experimental evaluation of Fall attacks. We describe the evaluation methodology, then present the results of the functional analyses, after which we present our evaluation of the key confirmation attack.

6.1 Methodology

We evaluated the effectiveness of FALL attacks on a set of ISCAS'85 benchmark circuits and combinational circuits from the Microelectronics Center of North Carolina (MCNC). Details of these circuits are shown in Table 1. These benchmark circuits remain reflective of contemporary combinational circuits and have been used extensively in prior work on logic locking, e.g. [15, 17, 21]. We implemented the TTLock and SFLL locking algorithms for varying values of the Hamming distance parameter h and maximum key size of 128 bits. Due to space limitations, we only show graphs/tables for the maximum key size of 64 bits. Results for the larger key size are discussed in the text in subsection 6.2. Locked netlists were optimized using ABC v1.01 [9] to minimize any structural bias introduced by our locking implementation.

ckt	**#in**	**#out**	**#keys**	**# of gates**		
				Original	**SFLL**	
					min	max
ex1010	10	10	10	2754	2783	2899
apex4	10	19	10	2886	2938	3058
c1908	33	25	33	414	1322	1376
c432	36	7	36	209	1119	1155
apex2	39	3	39	345	1367	1407
c1355	41	32	41	504	1729	1746
seq	41	35	41	1964	3177	3187
c499	41	32	41	400	1729	1750
k2	46	45	46	1474	2890	2903
c3540	50	22	50	1038	2591	2595
c880	60	26	60	327	2338	2368
dalu	75	16	64	1202	3284	3312
i9	88	63	64	591	2981	3015
i8	133	81	64	1725	3609	3637
c5315	178	123	64	1773	4076	4108
i4	192	6	64	246	2261	2289
i7	199	67	64	663	3038	3066
c7552	207	108	64	2074	4076	4105
c2670	233	140	64	717	2733	2775
des	256	245	64	3839	7229	7257

Table 1: Benchmark circuits. #in, #out and #key refer to the number of inputs, outputs and keys respectively.

6.1.1 Implementation

The circuit analyses were implemented in Python and use the Lingeling SAT Solver [4]. The key confirmation algorithm was implemented in C++ as a

modification to the open source SAT attack tool [6].

6.1.2 Execution Platform

Our experiments were conducted on the CentOS Linux distribution version 7.2 running on 28-core Intel® Xeon® Platinum 8180 ("SkyLake") Server CPUs. Although many opportunities for parallelization exist, our prototype implementation is single threaded. All algorithms were run with a time limit of 1000 seconds.

6.2 Circuit Analysis Results

Figure 3 show the performance of the circuit analyses attacks on the benchmarks in our experimental framework. Four graphs are shown: the left most of which is for SFLL-HD^0 while the remaining are for SFLL-HD^h with varying values of the Hamming Distance h. For each graph, the x-axis shows execution time while the y-axis shows the number of benchmark circuits decrypted within that time.

The **Distance2H attack defeats all SFLL-HD^h locked circuits for $h = \lfloor m/8 \rfloor$ and $h = \lfloor m/4 \rfloor$**. We repeated this experiment for **the seven largest circuits with a key size of 128 bits and the Distance2H attack defeated all of these locked circuits**. Recall that DISTANCE2H is not applicable when $4h > m$. ANALYZEUNATENESS is able to defeat 18 out of 20 SFLL-HD^0/TTLock circuits. SLIDINGWINDOW is able to defeat all locked circuits for $h = \lfloor m/8 \rfloor$, but does not perform as well for larger values of h. This is because the SAT calls for larger values of h are computationally harder as they involve more adder gates in the Hamming Distance computation. In summary, **65 out of 80 circuits (81%) are defeated** by at least one of our attack algorithms.

Among these 65 circuits for which the attack is successful, **a unique key is identified for 58 circuits (90%)**. This means **58 out of 80 circuits were defeated without oracle access** (I/O access to an unlocked IC) — only functional analysis of the netlist was required. Among the seven circuits for which multiple keys were shortlisted, the attack shortlists two keys which are bitwise complements of each other for four circuits, three keys are shortlisted for two other circuits. One corner cases occurs for c432: 36 keys are shortlisted, this is still a huge reduction from the initial space of 2^{36} possible keys.

6.3 Key Confirmation Results

Figure 4 shows the execution time of the key confirmation algorithm and compares and contrasts this with the "vanilla" SAT attack. Note that the y-axis is shown on a log scale. The bars represent the mean execution time of key confirmation for a particular circuit encoded with the various locking algorithms and parameters discussed above. Key values are obtained from the results of the experiments described in the previous subsection. The thin black line shows error bars corresponding to one standard deviation. We note that **key confirmation**

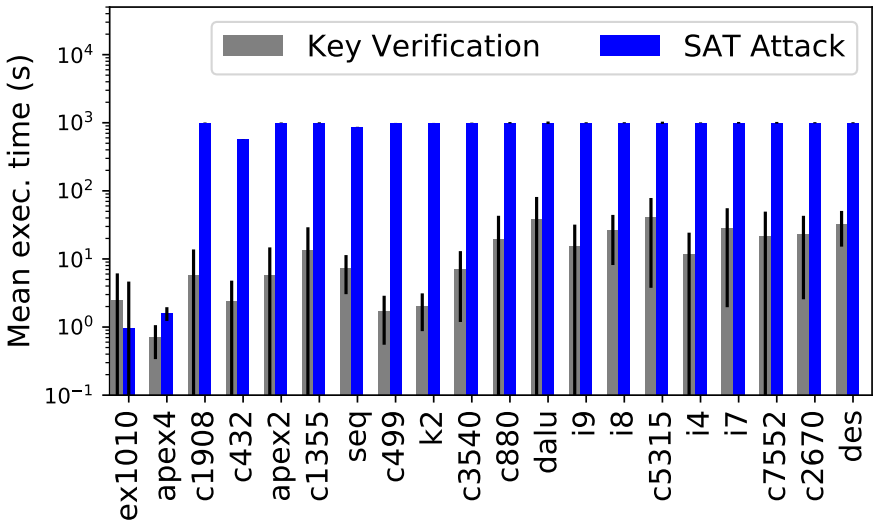

Figure 4: Mean exec. times of key confirmation and SAT attacks.

is orders of magnitude faster than the SAT attack while providing the same correctness guarantees.

Key confirmation provides a powerful new tool for attackers analyzing a locked netlist. Attackers can use some arbitrary circuit analysis to guess a few likely keys, and then use key confirmation to determine which (if any) of these is the correct key. **Key confirmation is applicable even if the locked netlist is SAT attack resilient**. Indeed, the SAT attack fails on most of these locked circuits as shown in Figure 3.

6.4 Discussion

Our results reinforce the observation that all logic locking schemes appear to be vulnerable to attack. We assert this is because the logic locking community has not adopted notions of provable security from cryptography. For instance, consider an adaptation of indistinguishability under chosen plaintext attacks (IND-CPA) [3] to logic locking. In this game, the defender initially picks two keys $\mathtt{K}_c^1$ and $\mathtt{K}_c^2$, and a bit $b \in \{0,1\}$. The game now proceeds in rounds. Each round consists of the adversary providing two different circuits to the defender. The defender encrypts one of them with $\mathtt{K}_c^b$. The adversary wins if they can guess which of the two circuits was encrypted with a non-negligible advantage over guessing. For SFLL-HD^h as the original circuit is largely unchanged by locking, so the adversary can win the game easily using any algorithm for circuit equivalence. In fact, to the best of our knowledge, the adversary would win the game described above for *all* logic locking schemes proposed so far. Truly

secure logic locking will need the development of a methodology that can win this game.

7 Conclusion

This paper proposed a set of Functional Analysis attacks on Logic Locking (Fall attacks). We developed structural and functional analyses to determine potential key values of a locked logic circuit. We then showed how these potential key values could be verified using our key confirmation algorithm.

Our work has three important implications. First, we showed how arbitrary structural and functional analyses can be synergistically combined with powerful Boolean reasoning engines using the key confirmation algorithm. Second, our attack was shown to often succeed (90% of successful attempts in our experiments) without requiring oracle access to an unlocked circuit. This suggests that logic locking attacks may be much more easily carried out than was previous assumed. Finally, the Fall attack successfully defeated secure function logic locking (SFLL), the only locking algorithm resilient to known attacks on logic locking. Experiments showed that Fall defeated 65 out of 80 benchmark circuits locked using SFLL.

References

[1] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121 – 123, 1979.

[2] A. Baumgarten, A. Tyagi, and J. Zambreno. Preventing IC Piracy Using Reconfigurable Logic Barriers. *IEEE Design and Test*, 27(1), Jan 2010.

[3] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A Concrete Security Treatment of Symmetric Encryption. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*. IEEE, 1997.

[4] A. Biere. Lingeling, Plingeling and Treengeling. In A. Balint, A. Belov, M. Heule, and M. Järvisalo, editors, *Proceedings of the SAT Competition*, 2013.

[5] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[6] `https://bitbucket.org/spramod/host15-logic-encryption`, 2015.

[7] Defense Science Board Task Force on High Performance Microchip Supply. `http://www.acq.osd.mil/dsb/reports/ADA435563.pdf`, 2005.

[8] IHS Technology Press Release: Top 5 most counterfeited parts represent a $169 billion potential challenge for global semiconductor industry. https://technology.ihs.com/405654/top-5-most-counterfeited-parts-represent-a-169-billion-potential-challenge-for-global-semiconductor-market, 2012.

[9] Alan Mishchenko. ABC: System for Sequential Logic Synthesis and Formal Verification. https://github.com/berkeley-abc/abc, 2018.

[10] M. Pecht and S. Tiku. Bogus! Electronic manufacturing and consumers confront a rising tide of counterfeit electronics. *IEEE Spectrum*, May 2006.

[11] S.M. Plaza and I.L. Markov. Solving the third-shift problem in ic piracy with test-aware logic locking. In *IEEE Transactions on CAD of Integrated Circuits and Systems*, 2015.

[12] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri. Security Analysis of Logic Obfuscation. In *Proceedings of the Design Automation Conference*, 2012.

[13] J. A. Roy, F. Koushanfar, and I. L. Markov. EPIC: Ending Piracy of Integrated Circuits. In *Proceedings of Design, Automation and Test in Europe*, 2008.

[14] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin. Appsat: Approximately deobfuscating integrated circuits. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2017.

[15] Yuanqi Shen and Hai Zhou. Double DIP: Re-Evaluating Security of Logic Encryption Algorithms. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, 2017.

[16] Semiconductor Industry Association: Anti-Counterfeiting Whitepaper One-Pager. http://www.semiconductors.org/clientuploads/directory/DocumentSIA/Anti%20Counterfeiting%20Task%20Force/ACTF%20Whitepaper%20Counterfeit%20One%20Pager%20Final.pdf, 2013.

[17] P. Subramanyan, S. Ray, and S. Malik. Evaluating the Security Logic Encryption Algorithms. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2015.

[18] R. Torrance and D. James. The State-of-the-Art in IC Reverse Engineering. In *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, 2009.

[19] J. Villasenor and M. Tehranipoor. The Hidden Dangers of Chop-Shop Electronics. *IEEE Spectrum*, Sep 2013.

[20] Y. Xie and A. Srivastava. Mitigating SAT Attack on Logic Locking. In *International Conference on Cryptographic Hardware and Embedded Systems*, 2016.

[21] Y. Xie and A. Srivastava. Anti-sat: Mitigating sat attack on logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[22] X. Xu, B. Shakya, M.M Tehranipoor, and D. Forte. Novel Bypass Attack and BDD-based Tradeoff Analysis Against all Known Logic Locking Attacks. In *Cryptology ePrint Archive*, 2017.

[23] M. Yasin, B. Mazumdar, S.S. Ali, and Sinanoglu O. Security Analysis of Logic Encryption against the Most Effective Side-Channel Attack: DPA. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, 2015.

[24] M. Yasin, B. Mazumdar, J. J. V. Rajendran, and O. Sinanoglu. SARLock: SAT attack resistant logic locking. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 236–241, 2016.

[25] M. Yasin, B. Mazumdar, O. Sinanoglu, and Rajendran J. Removal Attackson Logic Locking and Camouflaging Techniques. In *IEEE Transactions on Emerging Topics in Computing*, 2017.

[26] M. Yasin, S.M. Saeed, J. Rajendran, and O. Sinanoglu. Activation of logic encrypted chips: Pre-test or post-test? In *Design, Automation Test in Europe.*, 2016.

[27] Muhammad Yasin, Abhrajit Sengupta, Mohammed Thari Nabeel, Mohammed Ashraf, Jeyavijayan (JV) Rajendran, and Ozgur Sinanoglu. Provably-secure logic locking: From theory to practice. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, 2017.

[28] Muhammad Yasin, Abhrajit Sengupta, Benjamin Carrion Schafer, Yiorgos Makris, Ozgur Sinanoglu, and Jeyavijayan (JV) Rajendran. What to lock?: Functional and parametric locking. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, 2017.