Is Register Transfer Level Locking Secure?

Red Team: Chandan Karfa* and Ramanuj Chouksey*
Blue Team: Christian Pilato†, Siddharth Garg‡ and Ramesh Karri‡
*Indian Institute of Technology Guwahati, India
†Politecnico di Milano, Italy
‡New York University, USA
*{ckarfa, r.chouksey}@iitg.ac.in, †christian.pilato@polimi.it, ‡{siddharth.garg, rkarri}@nyu.edu

***Abstract*—Register Transfer Level (RTL) locking seeks to prevent intellectual property (IP) theft of a design by locking the RTL description that functions correctly on the application of a key. This paper evaluates the security of a state-of-the-art RTL locking scheme using a satisfiability modulo theories (SMT) based algorithm to retrieve the secret key. The attack first obtains the high-level behavior of the locked RTL, and then use an SMT based formulation to find so-called distinguishing input patterns (DIP)[1]. The attack methodology has two main advantages over the gate-level attacks. First, since the attack handles the design at the RTL, the method scales to large designs. Second, the attack does not apply separate unlocking strategies for the combinational and sequential parts of a design; it handles both styles via a unifying abstraction. We demonstrate the attack on locked RTL generated by TAO [1], a state-of-the-art RTL locking solution. Empirical results show that we can partially or completely break designs locked by TAO.**

I. Introduction

Many semiconductor companies are *fabless*, i.e., they use offshore third-party foundries to manufacture their chips [2]. While cost effective, the fabless model introduces security concerns. Since the foundry has access to the chip layout, it can reverse engineer the chip's functionality and steal the designer's intellectual property (IP). IP theft of this nature is a serious concern. One approach to preventing IP piracy is *logic locking* [3]–[5]. In this approach, the circuit functionality is locked using an additional input, called the *key*. Various internal signals of the IC are gated with bits of the key. The IC only functions correctly for a secret key value, known only to the designer, and otherwise produces corrupted outputs. When fabricated chips are received from the foundry (note, the foundry does not know the secret key), the designer activates the chip by loading the correct key in a tamper-proof memory.

Starting with the SAT attack [6], the past few years have witnessed a flurry of activity on logic locking, both on the attack and defense side. While these attacks and subsequent defenses are summarized in Section III, we note that a provably secure defense against the original SAT attack is still missing. In the SAT attack, the foundry has access to the locked netlist (at the gate-level) and a functioning chip purchased from the market. The attacker then uses the input/output behaviour of the functioning chip along with a SAT solver to infer the correct key. First published for breaking combinational circuits, the SAT attack has since been applied to sequential circuits as well [7]–[9]. However, since the attack operates at the gate-level, these techniques are not scalable to practical designs with hundreds of thousands of gates and flip-flops.

Recent work has advocated for defenses that perform logic locking during high-level synthesis (HLS); the resulting RTL locked netlists are *large* and consequently less vulnerable to conventional gate-level SAT attacks. To defeat such RTL locking mechanisms, an attack that works at higher levels of abstraction is desirable. The research question that we attempt to answer is: *"Can one scale the SAT attack to locked RTL?"*

We propose a satisfiability modulo theories (SMT) based algorithm to determine the secret key of a locked RTL design. The algorithm models an RTL design as a RTL finite state machine with datapath (RTL-FSMD) by applying the rewriting approach in [10]. We abstract out the details of the hardware into a behavioral program on which we launch an SMT based attack. Our attack finds distinguishing input patterns (DIPs) iteratively (similar to [6]) to rule out equivalence classes of incorrect keys and stops when no DIPs are found.

For linear arithmetic with m component keys[2], our algorithm is guaranteed to stop within m iterations. Our method works even for non-linear arithmetic since this is supported by the state-of-the-art Z3 SMT solver [11]. Further, our algorithm works on sequential circuits since the analysis is performed on an algorithmic abstraction of the design. We show that the locking keys inserted by TAO [1] can be recovered on HLS benchmarks. To the best of our knowledge, this is the *first* attack on RTL locking.

The paper is organized as follows. Preliminary concepts, including the attack model, the TAO locking approach and the FSMD model, are given in Section II. Section III discusses existing approaches to attack gate-level locking. Our attack/unlocking algorithm is given in Section IV. Section V presents the experimental methodology, results and limitations.

II. Background

This section presents the background required to understand the SMT-based attack.

A. Attack Model

We assume a malicious foundry that wishes to steal the RTL IP. To protect against this threat, we assume that the designer

[1]i.e., inputs that help eliminate incorrect keys from the keyspace.

[2]The actual key size is proportional to m.

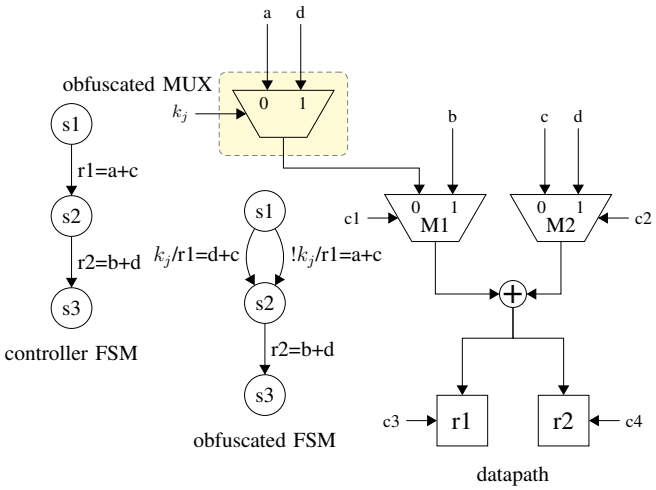

Fig. 1: An example of TAO obfuscation.

uses an RTL locking tool like TAO to produce locked RTL[3], performs synthesis and physical design and sends the layout of the locked design to the foundry. As in prior work, we assume the foundry is able to extract the gate-level netlist of the locked chip from its layout. Further, using techniques proposed in [12], we assume the foundry extracts RTL descriptions of the datapath and controller from the gate-level description of the locked chip. Finally, the foundry purchases a functioning (unlocked) copy of the chip from the market and can apply inputs to the chip and observe corresponding outputs (this is the oracle chip). Using this setup, the foundry attempts to recover the secret key to obtain the correct RTL.

B. TAO: An RTL Locking Approach

TAO [1] is an algorithm-level locking technique that applies during high-level synthesis (HLS). TAO hides selected constants, control branches and datapath operations based on an input locking key K. The key K is provided by the designer through an additional port to the design and partitioned into sub-keys used for each element to lock. The circuit will work correctly only when the correct locking key is given.

1) Constant locking: TAO identifies all the constants in the input behavior. It assumes a predefined-number of bits x to implement all constants. Each constant c_i^p of the behavior is locked as $c_i^e = c_i^p \oplus k_i$, where c_i^e is the locked value stored in hardware and k_i is a x-bit key. The correct constant can be obtained by reversing the operation $c_i^p = c_i^e \oplus k_i$.

2) Branch Locking: Each branch in the input behavior (and hence in the controller FSM) is locked with a one bit key. If the condition $c_p == 1$ is checked in a control state, the condition is modified as $c_p \oplus k_j == 1$, where k_j is a one bit key. k_j is part of the locking key K and locks this condition checking. The right branch is taken by the controller with the correct k_j.

[3]Note that TAO performs locking during high-level synthesis and outputs locked RTL with separate datapath and controller.

3) Datapath Locking: TAO adds decoy multiplexer-based interconnections between registers and the functional units. Each MUX is controlled by a key bit k_l. The correct output is connected to 0 or 1 input of this MUX based on the correct value of k_l. This MUX multiplexes the correct and the spurious data flow in each control state. Only with the correct key, the correct operations are performed.

C. FSMD Model

An FSMD is a specification model that can represent all hardware designs and is defined as a 7-tuple $\langle Q, q_0, I, V, O, f, h\rangle$, where Q is the finite set of states, $q_0 \in Q$ is the reset (initial) state, I is the finite set of inputs, V is the finite set of storage variables, O is the set of outputs, $f : Q \times 2^S \rightarrow Q$ is the state transition function, $h : Q \times 2^S \rightarrow U$ is the update function of the output and the storage variables. Here S is a set of relations over arithmetic expressions and Boolean literals and U is a set of storage variables and output assignment statements. The FSMD is inherently deterministic [13]. In our attack implementation, we represent the FSMD model of the RTL design, referred to as an RTL-FSMD model. In RTL-FSMD, V has all the registers and the memories in the design. TAO generated locked RTL is converted into an RTL-FSMD using a *rewriting* (explained in Section IV-B). It also embeds the key values and describes how the behavior evolves with different key values.

Consider the design in Fig. 1. The operations $r1 = a + c$ and $r2 = b + d$ are performed in the datapth in states $s1$ and $s2$, respectively. The MUX in the yellow box is added to lock the first operation and is controlled by the key bit k_j. The correct key value is $k_j = 0$. Therefore, if $k_j = 1$ is supplied, $r1 = d + c$ will be executed producing a wrong result. The locked RTL behavior is shown in the locked FSM with an additional transition between $s1$ and $s2$. The key is implicit to the controller FSM. However, when the RTL-FSMD is reverse engineered from the layout, the key k_l in unknown and creates additional transitions in the RTL-FSMD.

III. Related Work

The paper builds on a body of work in hardware security on *logic locking*, an idea first proposed by [6]. However the SAT attack [14] was able to defeat all locking schemes at the time of its publication in a matter of minutes. However, the SAT attack was evaluated on small gate-level locked netlists. Subsequent attacks on logic locking have sought to scale the original SAT attack [15], [16], extend its applicability to sequential circuits [7]–[9], and use hints from the locked netlist's structure to recover the secret key [17], [18]. Several locking countermeasures have been proposed to thwart these attacks [4], [19]–[21]. However, none has yet been shown to be provably secure. These attacks and defenses focus on the gate-level abstraction and have been demonstrated on small circuits like the ISCAS benchmarks. Recently, there has been an attempt to perform logic lock at the RTL [1], [22] and at the C level [23]. TAO is an example of such a scheme. However, to the best of our knowledge, the security of RTL locking has

not been studied; the proposed SMT attack is the first one on RTL locking. While SMT has been used to unlock gate-level netlists [24], these methods do not apply to RTL unlocking.

IV. Attack Methodology

A. Problem Formulation

The objective is to find the locking key K using an SMT solver and by querying an activated IC (the Oracle).

The RTL-FSMD $P(I, O, K) \in \mathbb{Z}^{M+N+K}$ has M primary inputs, N primary outputs and K unknown keys. It represents the input/output relation of the locked RTL design based on the key. $C_O = (I, O)$ is the input/output relation of the activated IC. The attacker can apply inputs to $C_O \in \mathbb{Z}^{M+N}$ and observe the correct output. However, the attacker cannot model the internal behavior of C_O, a *black-box* function $eval(X_i) = Y_i$. For an input X_i, $eval(X_i) = Y_i$ *iff* $C_O(X_i, Y_i)$. While we assume that all inputs/outputs are Integer, this formulation works for Real numbers.

As shown in Fig. 2, the RTL-FSMD consists of a set of states and transitions among the states which represent the control flow. Each transition is associated with a condition and a set of operations that execute in parallel. The data dependencies among the operations represent the data flow. We unroll each loop and the RTL-FSMD is thus a directed acyclic graph. The RTL-FSMD has a *start/reset state* from which any execution starts and terminates. We assume the behavior is deterministic. A *trace* in an RTL-FSMD represents a path from the *reset state* back to the *reset state*. For a trace τ, the *condition of execution* C_τ over $I \cup C \cup K$, where I is the set of inputs, C is set of integer constants and K is the set of unknown keys, represents the symbolic condition that must be satisfied by the initial data state to execute the trace. The C_τ is the weakest precondition of the the trace τ [25]. The *data transformation* D_τ of τ is an ordered tuple of algebraic expressions $\langle e_j \rangle$ over $I \cup C \cup K$ such that e_j represents the value of the output $o_j \in O$ after execution of the trace. C_τ and D_τ can be obtained by the symbolic execution of the trace [13].

The RTL-FSMD consists of a finite set of traces $\{\tau_1, \tau_2, \ldots, \tau_k\}$. The output of an RTL-FSMD will be obtained by the execution of one trace depending on the input values. Each trace has a non-overlapping condition of execution since the behavior is deterministic. Therefore, the outputs O in the RTL-FSMD can be represented as $P(I, O, K) : \; O = (ite\ C_{\tau_1} D_{\tau_1}\ (ite\ C_{\tau_2} D_{\tau_2}\ (ite\ \ldots (ite\ C_{\tau_{k-1}} D_{\tau_{k-1}} D_{\tau_k})) \ldots))$ where $(ite\ C\ D_1\ D_2)$ (aka if-then-else) indicates if the condition C is `True` return the value D_1 else D_2. For a given input I_i and a key K_l and corresponding output O_j, the execution of the P is $P(I_i, O_j, K_l)$. The trace τ_x is executed for this input and key combination, i.e., C_{τ_x} is evaluated to `True` for I_i and K_l. Therefore, $P(I_i, O_j, K_l)$ represents the transformation D_{τ_x} of τ_x, i.e., $P(I_i, O_j, K_l) = D_{\tau_x}$.

B. Rewriting Method

The HLS-generated RTL consists of a datapath and a controller FSM [26]. In each transition in the FSM, control signals are assigned with value 0/1. Our objective is to identify the corresponding RTL operations performed in the datapath. The control signal assignments in each controller FSM are replaced with corresponding RTL operations. This way, the datapath and the controller details are abstracted out and we have a RT-level behaviour. The concept is explained in Fig. 2. To obtain the RTL operations in each state, we extend the rewriting method presented in [10] as discussed below.

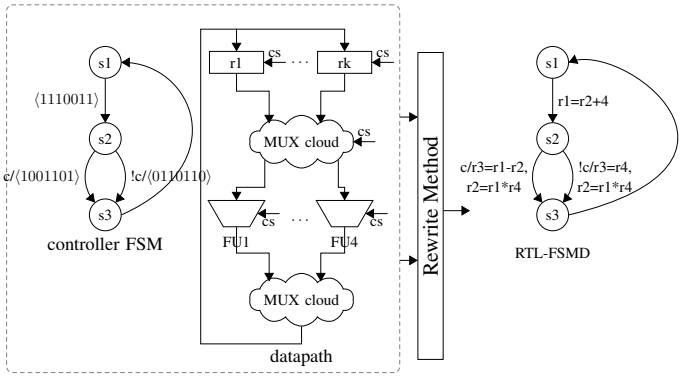

Fig. 2: RTL-FSMD from RTL using rewriting approach.

In the datapath, signal flow is controlled by the control signals. For each datapath module, input $\rightarrow$ output assignments are termed as micro-operations. For example, for a multiplexer $out = MUX(in1, in2, sel)$, there are two possible micro-operations, i.e., $out \leftarrow in1$ and $out \leftarrow in2$ and the associated control signal assertions are $sel = 0$ and $sel = 1$, respectively. Given a control signal assignment, we can identify the active micro-operations due this control signal assignment. A micro-operation not associated with any control signal is always active. The *rewriting method* identifies the *spatial sequence* of data flow needed for an RTL operation in a reverse order. The method consists in rewriting terms one after another in an expression. The micro-operations of the form $r \leftarrow r_{in}$ in which a register occurs in the left-hand side (LHS) are found first. Next, the right-hand side (RHS) expression r_{in} is rewritten by looking for an active micro-operation $r_{in} \leftarrow s$ or $r_{in} \leftarrow s1 < op > s2$. Next, s ($s1$ or $s2$ in the latter case) are rewritten provided they are not registers. The rewriting takes place from left to right in a breadth-first manner and terminates when all signals in the RHS expression are registers.

C. Algorithm Description

The problem of finding the distinguishing input pattern (DIP) can be modelled as follows: *Given two key values K_1 and K_2 and an input I^d, the output obtained is O_1 and O_2, respectively. The input I^d is DIP for K_1 and K_2 iff* $P(I^d, O_1, K_1) \wedge P(I^d, O_2, K_2) \wedge (O_1 \neq O_2)$. Once a DIP is found, the output is obtained from the activated IC. The DIP formulation is strengthened by adding this input/output relation for both K_1 and K_2. This process repeats in an iterative manner until no DIP found. In this time, we will check the SAT of the DIP formula with $(O_1 \equiv O_2)$. Any

Algorithm 1: Algorithm to recover the keys.

Input : P, eval
Output: The values of K

1 i = 1 ;
2 $F_1 = P(I, O_1, K_1) \wedge P(I, O_2, K_2)$;
3 **while** $sat[F_i \wedge (Y_1 \neq Y_2)]$ **do**
4 $\quad I_i^d$ = a DIP value that satisfy $[F_i \wedge (Y_1 \neq Y_2)]$;
5 $\quad O_i^d = eval(I^d)$;
6 $\quad F_{i+1} = F_i \wedge P(I_i^d, O_i^d, K_1) \wedge P(I_i^d, O_i^d, K_2)$;
7 $\quad i = i + 1$;
8 **end while**
9 K = the value of K in the sat assignment of $F_i \wedge (Y_1 \equiv Y_2)$;

assignment of K_1 or K_2 for this formula is the correct key. One can recover K using Algorithm 1.

Algorithm 1 always terminates. The formula $P(I_i^d, O_i^d, K_1)$ is an equation linking the unknown keys. So, we add an equation relating the unknown keys in each iteration. Each iteration gets a DIP that rules out one incorrect equivalence classes of keys. Therefore, the equation from each iteration results in an independent equation. If P involves linear arithmetic, one can obtain the values of the K unknown variables by solving the K independent equations connecting them. So, Algorithm 1 finishes in $\|K\|$ steps for linear arithmetic. For non-linear arithmetic, the algorithm resolves when sufficient equations are set up. The search space reduces in each iteration. Therefore, the algorithm completes in a finite number of iterations. The key recovered is consistent with all the observed input/output patterns and thus represents the correct key.

D. An Illustrative Example

In the locked RTL code in Listing 1, two constants are locked with $k1$ and $k2$. Moreover, the condition is locked with a Boolean variable $c1$. Assume that $k1 = 5, k2 = 3, c1 = False$ in the original program. Our objective is to recover these values from the locked RTL with the help of an oracle.

Listing 1: if-else block

```
c = a > b
if(c xor c1)
  out = a + k1
else
  out = b * k2
```

Consider the SMT code in Listing 2. The function A in this SMT code models the functionality of the behavior in Listing 1. The SMT code to obtain DIP is given by the next three assert statements. *"Does there exist an assignment of a and b such that for two different values of $k1$ (i.e., $k11$ and $k12$) and $k2$ (i.e., $k21$ and $k22$), we have two different outputs?"*. Z3 returns $a = 1, b = 1$ and the corresponding output is 3.

The assertions added in iteration 2 are shown in the first part of the Listing 3. The process continues for three more iterations and the assertions added into the DIP model are shown in the rest of Listing 3. In the 4th iteration, Z3 returns $UNSAT$. We obtain $k1 = 5, k2 = 3, c1 = False$ by reversing SAT (i.e., $(assert\ (= out1\ out2))$) as correct keys.

Listing 2: SMT code to obtain the DIP for Listing 1

```
(declare-const a Int)
(declare-const b Int)
(declare-const k11 Int)
(declare-const k21 Int)
(declare-const k12 Int)
(declare-const k22 Int)
(declare-const out1 Int)
(declare-const out2 Int)
(declare-const c1 Bool)
(declare-const c2 Bool)
(define-fun G ((a Int) (b Int)) Bool (> a b))
(define-fun A ((a Int) (b Int) (x1 Int) (x2 Int)
               (x3 Bool) (c Bool)) Int (ite
               (xor c x3) (+ a x1) (* b x2)))
(assert (= out1 (A a b k11 k21 c1 (G a b))))
(assert (= out2 (A a b k12 k22 c2 (G a b))))
(assert (not (= out1 out2)))
(check-sat)
(get-model)
```

Listing 3: Assertion refinements in successive iterations

```
;Iteration 2: a = 0, b = 0 → out = 0
;added assertions
(assert (= 0 (A 0 0 k11 k21 c1 (G 0 0))))
(assert (= 0 (A 0 0 k12 k22 c2 (G 0 0))))
Iteration 3: a = -5, b = -1 → out = -3
;added assertions
(assert (= -3 (A -5 -1 k11 k21 c1 (G -5 -1))))
(assert (= -3 (A -5 -1 k12 k22 c2 (G -5 -1))))
Iteration 4: a = -3, b = -4 → out = 2
;added assertions
(assert (= 2 (A -3 -4 k11 k21 c1 (G -3 -4))))
(assert (= 2 (A -3 -4 k12 k22 c2 (G -3 -4))))
```

E. Attack Tool-flow

Fig. 3 is our implementation flow. The tool parses the locked RTL generated by TAO using Pyverilog [27] (RTL→ FSMD module). It uses a rewriting method yielding an RTL-FSMD [10] and transforms the RTL-FSMD to feed into the KLEE tool [28] to get the symbolic representation of the outputs as discussed in section IV-A. This symbolic representation of the program creates the SAT formulation for DIP. It invokes the SMT tool Z3 [11]. If Z3 cannot prove the SAT/UNSAT of the formula in any iteration, our algorithm fails. If Z3 returns SAT, the corresponding inputs are used to get the correct output using the functional IP. It strengthens the DIP formula with this input/output relation and it calls Z3 again. The algorithm unlocks the keys once Z3 returns UNSAT. The tool flow is automated. RTL → FSMD module is in Python and we write the rest of the tool flow in C++.

We invoke SMT solver Z3 [11] to check for Satisfiability on line 3 of Algorithm 1. SMT solvers require the programs to be in static single assignment (SSA) [29] form. In the SSA form, each variable is assigned exactly once. We model the RTL-FSMD as a formula consisting of the condition of executions and the data transformations of all the traces. This formula represents the one time assignment of each output. So, it is already in the SSA form. This formula is computed using KLEE [28] even if it is symbolic technique that does not require the program to be in SSA form.

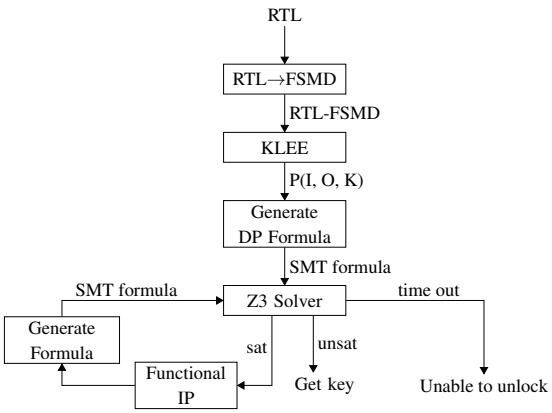

Fig. 3: Outline of the SMT based unlocking of TAO.

V. Empirical Evaluation

A. Methodology

To evaluate our methodology, we emulated a red team-blue team activity in our experiments. The two teams are in separate institutions. We use three HLS benchmarks - WAKA, ARF and Motion for our experiments. A blue team designer (not in the red team institution) synthesized these benchmarks with TAO to generate the locked RTL in Verilog [1]. For each test scenario, the number of lines in the Verilog code, the number of multiplications, additions and subtractions in the locked RTL are reported in the columns 2, 3, 4 and 5, respectively, in Table I. As discussed in the section II-B, TAO applies operation, control and constant locking. The amount of each type of locking is controlled by input parameters. Using these parameters, For each benchmark, the blue team generated several locked designs with differing operations, control-flow statements and constants obfuscated, as shown in columns 6, 7, 8 of Table I, resulting in different key sizes of up to 155 bits, as shown in column 9 of the same table. To check the size of the gate-level circuits targeted by our approach, we synthesized the RTL using Synopsys Design Compiler targeting the SAED 32nm technology. We note that the designs are large with up to 14K combinational cells and 3K sequential cells, as reported in columns 10 and 11, respectively, of Table I.

The red team uses the methodology in this paper to unlock the designs. The red team unlocking results are tabulated in columns 12-15 of Table I. This includes the number of iterations (Ite) of Algorithm 1 to unlock the key, the number of instructions (Ins) processed by KLEE and the CPU time (Time) and the memory usage (RAM) for each test case. For these experiments, we use Z3 SMT solver version 4.8.5 - 64 bit, with a time out of 10 hours. As shown in Table I, our unlocking algorithm recovers the keys in a few iterations. For successful cases, the *time to unlock is under 30 minutes*. For three cases, Z3 solver times out after a few iterations. We discuss these scenarios in the next section. None of the previously reported combinational unlocking techniques [4], [6], [19]–[21] apply in our setting since our locked netlists are sequential. On the other hand, the gate-level SAT attacks on sequential circuits [7]–[9] reported results for ISCAS'89 and ITC'99 benchmarks, while we scale to much larger benchmarks.

TABLE I: Results: Unlocking TAO-locked RTL designs.

Bench	LOC	×	+	-	Operations	Conditions	Constants	Key	Comb	Seq	Iterations	Instructions	Time (s)	RAM (MB)
WAKA	753	-	13	7	-	-	3	65	1255	917	4	524	5.16	28
	779	-	23	11	11	4	-	11			5	653	35.46	43
	773	-	23	11	11			9			4	617	92.39	40
	828	-	21	9	9	4	3	73			45	672	1157.13	138
ARF	1431	21	27	10	-	6	-	3	19715	3381	2	6185	517.80	661
	1654	21	27	10	-	-	1	32			2	6863	406.97	576
	1647	21	65	34	65			32			5	6718	>10hrs	-
MOTION	1140	19	11	0	-	-	2	64	13938	2924	5	931	7.01	16
	1239	15	29	10	37	-	-	27			2	885	>10hrs	-
	1250	15	32	10	37	-	4	155			5	924	>10hrs	-

LOC: # of lines in obfuscated Verilog RTL. ×: # of multiplications in Verilog RTL. +: # of adds in Verilog RTL. -: # of subtracts in Verilog RTL. Operations: # of operations obfuscated. Conditions: # of conditions obfuscated. Constants: # of constants obfuscated. Key: # of key bits. Comb: # of combinational cells. Seq: # of sequential cells. Iterations: # of iterations. Instructions: # of instructions executed by KLEE.

TABLE II: Results: Unlocking a locked C code.

Bench	Operations	Conditions	Constants	key	Iterations	Instructions	Time (s)	RAM (MB)
WAKA	1	1	5	162	6	306	2888.91	92
	-	-	7	224	8	298	2658.56	120
	2	1	6	195	6	345	3495.51	98
ARF	2	1	1	35	3	1060	1579.77	861
	-	-	4	128	2	1068	400.77	718
	2	1	2	67	3	1142	>10hrs	-
MOTION	2	-	2	66	4	326	11.74	18
	6	-	6	198	8	421	>10hrs	-

Our approach is not limited to HLS-generated designs. It can work on locked C code. For example, in [23], a locked C code is given to a cloud HLS tool to avoid stealing the algorithm IP. To show that we can attack a locked C code, we created several C variants with a large number of key bits and report the results in Table II. For WAKA, we can unlock all cases in one hour. The biggest key that we unlocked is 224 bits. For ARF and Motion, we can unlock up to 128 and 66 bits keys, respectively. For larger key sizes, Z3 times out necessitating scalable approaches.

B. Discussion of the Results

Handling Time-outs: Solving SMT for arbitrary, non-linear arithmetic over the reals is undecidable [30]. Thus, the SMT solver may not prove the satisfiability of an formula comprising non-linear arithmetic. The SMT solver stops with an *unknown result* or *times out*. Although we did not come across

the *unknown* case, we encountered *time outs* for five instances (see Table I and Table II). We suspect that a time-out implies that no more DIPs exist, i.e., the attack has terminated although Z3 is unable to prove this formally. To substantiate this, we negate the formula (see step 9 of Algorithm 1) and Z3 returns the *correct* key in all five instances. Thus, even in the few cases that the attack times out, it yields a correct key.

Limitations: We did not implement extraction of arrays from Block RAM in RTL in the RTL $\rightarrow$ FSMD module yet. Also, functions in the input C code of TAO are in-lined before RTL generation. We will enhance our implementation to support these two features. This will help experiment on larger test cases.

VI. Conclusions and Future Work

This work presents an SMT attack to recover the secret key from a locked RTL netlist generated using the TAO RTL locking tool. Compared to gate-level attacks on sequential logic locking, the SMT attack abstracts all hardware details into a behavioral program, scaling to large designs. The attack is evaluated using a blue team-red team approach, wherein the blue team uses the TAO RTL locking tool to generate locked Verilog RTL along with the executable generated from input C code as an oracle to the red team. The red team unlocked large designs with up to 3K sequential cells and 195 key bits demonstrating the effectiveness of the attack.

Our future work will take two directions. First, we will leverage insights from our attack to strengthen RTL locking against SMT attacks. Our experiments indicate that non-linear equations are harder to solve and this could inform defenses. Second, we will scale the attacks to target larger designs.

VII. Acknowledgments

C. Karfa was partially supported by the DST, Government of India under Project ECR/2017/000492. R. Karri is funded in part by National Science Foundation grant number 1526405, ONR award number N00014-18-1-2058 and by NYU CCS and NYU-AD CCS-AD. S. Garg was supported in part by an NSF CAREER Award and NSF Grant 1527072.

References

[1] C. Pilato, F. Regazzoni, R. Karri, and S. Garg, "TAO: Techniques for algorithm-level obfuscation during high-level synthesis," in *IEEE/ACM Design Automation Conference*, June 2018, pp. 1–6.

[2] S. Heck, S. Kaza, and D. Pinner, "Creating Value in the Semiconductor Industry," accessed May 27, 2019. [Online]. Available: http://www.edn.com/design/integrated-circuit-design/4375454/Is-high-level-synthesis-ready-for-prime-time

[3] P. Tuyls, G.-J. Schrijen, B. Škorić, J. van Geloven, N. Verhaegh, and R. Wolters, "Read-proof hardware from protective coatings," in *CHES'06*, ser. LNCS, vol. 4249, 2006, pp. 369–383.

[4] M. Yasin, B. Mazumdar, J. J. V. Rajendran, and O. Sinanoglu, "Sarlock: Sat attack resistant logic locking," in *HOST'16*, May 2016, pp. 236–241.

[5] J. Rajendran, H. Zhang, C. Zhang, G. S. Rose, Y. Pino, O. Sinanoglu, and R. Karri, "Fault analysis-based logic encryption," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 410–424, Feb 2015.

[6] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the security of logic encryption algorithms," in *HOST'15*, May 2015, pp. 137–143.

[7] M. E. Massad, S. Garg, and M. Tripunitara, "Reverse engineering camouflaged sequential circuits without scan access," in *IEEE/ACM International Conference on CAD*, Nov 2017, pp. 33–40.

[8] Y. Kasarabada, S. Chen, and R. Vemuri, "On SAT-based attacks on encrypted sequential logic circuits," in *ISQED*, 2019, pp. 204–211.

[9] T. Meade, Z. Zhao, S. Zhang, D. Pan, and Y. Jin, "Revisit sequential logic obfuscation: Attacks and defenses," in *ISCAS*, May 2017, pp. 1–4.

[10] C. Karfa, D. Sarkar and C. Mandal, "Verification of Datapath and Controller Generation Phase in High-Level Synthesis of Digital Circuits," *IEEE TCAD*, vol. 29, no. f3, pp. 479–492, Mar 2010.

[11] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, ser. LNCS, vol. 4963, 2008, pp. 337–340.

[12] J. Rajendran, A. Ali, O. Sinanoglu, and R. Karri, "Belling the cad: Toward security-centric electronic system design," *IEEE TCAD*, vol. 34, no. 11, pp. 1756–1769, Nov 2015.

[13] C. Karfa, D. Sarkar, C. Mandal and P. Kumar, "An Equivalence-Checking Method for Scheduling Verification in High-Level Synthesis," *IEEE TCAD*, vol. 27, no. 3, pp. 556–569, Mar 2008.

[14] J. A. Roy, F. Koushanfar, and I. L. Markov, "Epic: Ending piracy of integrated circuits," in *ACM Design, Automation and Test in Europe*, 2008, pp. 1069–1074.

[15] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, "Appsat: Approximately deobfuscating integrated circuits," in *HOST'17*, 2017, pp. 95–100.

[16] D. Liu, C. Yu, X. Zhang, and D. Holcomb, "Oracle-guided incremental sat solving to reverse engineer camouflaged logic circuits," in *DATE*, 2016, pp. 433–438.

[17] 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 *CHES'17*, vol. 10529, 2017, pp. 189–210.

[18] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran, "Removal attacks on logic locking and camouflaging techniques," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2019.

[19] Y. Xie and A. Srivastava, "Anti-sat: Mitigating sat attack on logic locking," *IEEE TCAD*, vol. 38, no. 2, pp. 199–207, Feb 2019.

[20] M. Yasin, A. Sengupta, B. C. Schafer, Y. Makris, O. Sinanoglu, and J. J. Rajendran, "What to lock?: Functional and parametric locking," in *IEEE Great Lakes Symposium on VLSI*, 2017, pp. 351–356.

[21] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. J. Rajendran, and O. Sinanoglu, "Provably-secure logic locking: From theory to practice," in *ACM Conference on CCS*, Nov 2017, pp. 1601–1618.

[22] Y. Lao and K. K. Parhi, "Obfuscating dsp circuits via high-level transformations," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 23, no. 5, pp. 819–830, 2014.

[23] H. Badier, J. L. Lann, P. Coussy, and G. Gogniat, "Transient key-based obfuscation for HLS in an untrusted cloud environment," in *DATE*, 2019, pp. 1118–1123.

[24] K. Z. Azar, H. M. Kamali, H. Homayoun, and A. Sasan, "Smt attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the sat attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 97–122, 2019.

[25] Z. Manna, *Mathematical Theory of Computation*. Tokyo: McGraw-Hill Kogakusha, 1974.

[26] C. Pilato, S. Garg, K. Wu, R. Karri, and F. Regazzoni, "Securing hardware accelerators: A new challenge for high-level synthesis," *IEEE Embedded Systems Letters*, vol. 10, no. 3, pp. 77–80, Sep. 2018.

[27] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog HDL," in *Applied Reconfigurable Computing*, ser. LNCS, vol. 9040, 2015, pp. 451–460.

[28] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, Dec 2008, pp. 209–224.

[29] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct 1991.

[30] P. Nuzzo, A. Puggelli, S. A. Seshia, and A. Sangiovanni-Vincentelli, "Calcs: Smt solving for non-linear convex constraints," in *Formal Methods in Computer Aided Design*, Oct 2010, pp. 71–79.