RANE: An Open-Source Formal De-obfuscation Attack for Reverse Engineering of Logic Encrypted Circuits

Shervin Roshanisefat[1], Hadi Mardani Kamali[1], Houman Homayoun[2], Avesta Sasan[1]

[1] George Mason University, Fairfax, VA, USA.
{sroshani,hmardani,asasan}@gmu.edu

[2] University of California, Davis, CA, USA.
hhomayoun@ucdavis.edu

ABSTRACT

To enable trust in the IC supply chain, logic locking as an IP protection technique received significant attention in recent years. Over the years, by utilizing Boolean satisfiability (SAT) solver and its derivations, many de-obfuscation attacks have undermined the security of logic locking. Nonetheless, all these attacks receive the inputs (locked circuits) in a very simplified format (`Bench` or remapped and translated `Verilog`) with many limitations. This raises the bar for the usage of the existing attacks for modeling and assessing new logic locking techniques, forcing the designers to undergo many troublesome translations and simplifications. This paper introduces the RANE Attack, an open-source CAD-based toolbox for evaluating the security of logic locking mechanisms that implement a unique interface to use formal verification tools without a need for any translation or simplification. The RANE attack not only performs better compared to the existing de-obfuscation attacks, but it can also receive the library-dependent logic-locked circuits with no limitation in written, elaborated, or synthesized standard HDL, such as `Verilog`. We evaluated the capability/performance of RANE on *FOUR* case studies, one is the first de-obfuscation attack model on FSM locking solutions (e.g., HARPOON) in which the key is not a static bit-vector but a sequence of input patterns.

CCS CONCEPTS

• **Security and privacy → Hardware attacks and countermeasures**; **Hardware reverse engineering**.

KEYWORDS

Logic Locking, De-obfuscation, Formal Verification

ACM Reference Format:

Shervin Roshanisefat, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. 2021. RANE: An Open-Source Formal De-obfuscation Attack for Reverse Engineering of Logic Encrypted Circuits. In *GLSVLSI '21: Great Lakes Symposium on VLSI, June 22–25, 2021, Virtual Event, USA. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3453688.3461760*

GLSVLSI '21, June 22–25, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8393-6/21/06…$15.00
https://doi.org/10.1145/3453688.3461760

1 INTRODUCTION

To save the ever-increasing costs of maintaining an integrated circuit (IC) supply chain facility, take advantage of cutting-edge technology nodes, and meet the market demand, the manufacturing supply chain of ICs is globally distributed [3]. However, in a horizontal (globally distributed) IC supply chain, due to the lack of trust and with no reliable monitoring, many security threats have emerged, including IC overproduction, Trojan insertion, reverse engineering, IP theft, and counterfeiting [34]. Amongst several *design-for-trust* (DfTr) techniques, logic obfuscation, *a.k.a.* logic locking, received significant attention in recent years [4, 11, 12, 17, 18, 21, 35, 36, 53]. Logic locking is the mechanism of concealing the functionality of the circuit using its secret. The correct functionality of the circuit will be ensured whenever the correct secret value is provided to the circuit. The secret could be provided by different means. It could be an explicit value statically loaded from tamper-proof memory (TPM) to the circuit at power ON referred to as the key (key-based logic locking) [41]. It also could be an implicit sequence of input patterns from primary inputs (PIs), which make the transition in the circuit's FSM to its desired state (key-less logic locking) [2, 16, 44].

Over the recent years, many de-obfuscation attacks subvert the trustworthiness of logic locking. In particular, *Boolean* satisfiability (SAT)-based attack and its derivatives show how well-formulated attack models could break the existing logic locking techniques [25] with fast convergence. Depending on the assumptions of the threat models used in de-obfuscation attacks, these attacks may be able to target the combinational parts of the circuit, separately through the design-for-test (DFT) infrastructure (i.e., scan chain pins) [6, 15, 20, 24, 27, 31, 40, 45, 52], or they have to target the sequential circuit as a whole (through PI/PO) [22, 28, 32, 37]. Nevertheless, many of these attacks have been developed over the basic capabilities of different solvers that have lots of limitations, making them less practical on real applications. For instance, almost all open-source de-obfuscation attack tools [22, 24, 40, 50] receive the locked circuit in `Bench` or *translated and remapped* `Verilog` format, converted by open-source synthesis tools like ABC and Yosys [5, 42]. So, many real applications with complex macros require a heavy library-dependent conversion and simplification before exploiting these attacks. In many cases, these attacks fail to evaluate the robustness of existing logic locking techniques in such scenarios.

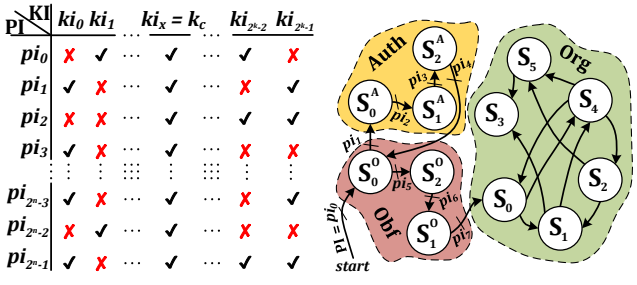
In this paper, we introduce a unified Reverse Assessment of Netlist Encryption (**RANE**), as an open-source CAD-based toolbox for evaluating the security of different logic locking techniques. Unlike the existing open-source de-obfuscation tools on logic locking, RANE has been developed based on a unified framework with a unique interface to exploit the capability/scalability of formal

verification tools for different stages of the attack. The RANE framework allows the users to exploit any formal verification tool, either open-source or commercial, such as Cadence JasperGold, Synopsys Formality, SymbiYosys, etc. The establishment of such formal verification tools allows RANE to support circuits written, elaborated, synthesized in standard HDL, like `Verilog`, with no limitation on the technology library used for the design/implementation. The flexibility and deployability of formal verification tools also allow us to model different threat models in RANE. With more concentration on sequential-based attacks on logic locking, in this paper, we will evaluate the capability/performance of RANE on *FOUR* different case studies: (1) an oracle-less attack on key-less (implicit key) sequence-based logic locking (HARPOON), (2) An oracle-guided attack on HARPOON, (3) An oracle-guided attack on sequential logic locking (scan is BLOCKED), and (4) An oracle-guided attack on combinational logic locking (scan is OPEN). We also demonstrate how the RANE framework could be extended to support any form of de-obfuscation attack that relied on formal verification tools.

2 BACKGROUND ON LOGIC LOCKING AND DE-OBFUSCATION ATTACKS

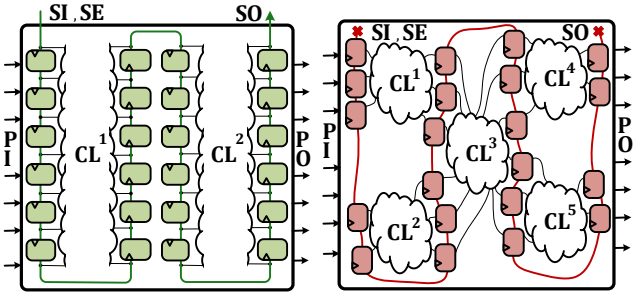
Logic locking is the process of hiding the functionality of a circuit by implementing post-manufacturing means of programmability into the netlist. Logic locking has been widely studied in the literature [4, 10–12, 17, 18, 21, 26, 35, 48, 53], in which, the functionality of a circuit is locked using two major techniques. It could be implemented as (1) a set of dedicated key inputs (mostly driven from TPM) such that only when the correct key is applied, the circuit resumes its expected (correct) functionality [4, 10–12, 17, 18, 21, 26, 35, 36, 46–48, 53], or (2) a set (sequence) of input patterns through PIs that requires to be traversed to lead the state of the circuit to the normal (correct) mode [2, 16, 44].

Fig. 1 depicts how both key-based and key-less logic locking techniques work. In key-based logic locking shown in Fig. 1(a)), there exist two sets of inputs, i.e., primary inputs (PI) and key inputs (KI). In key-based logic locking, the key values are stored and initiated in TPM. However, after reverse-engineering, the content of TPM will be wiped out, and the adversary has to evaluate them as extra inputs to the circuit (KI). In key-less logic locking, on the other hand, as depicted in Fig. 1(b), new state modes, such as *obfuscation* mode and *authentication* mode, are added in the circuit's FSM required to be traversed using a specific set of PI patterns. For example, by applying patterns as $pi_1 \rightarrow pi_7$ to the PI of 1(b), the circuit will reach its normal (correct) initial state. In this case, there exists no extra (dedicated) wires/inputs as the KI, which makes the attack formulation much harder for the adversary.



(a) Key-based Example (b) HARPOON: Key-less

Figure 1: Explicit Key-based vs. Implicit Sequence-based.



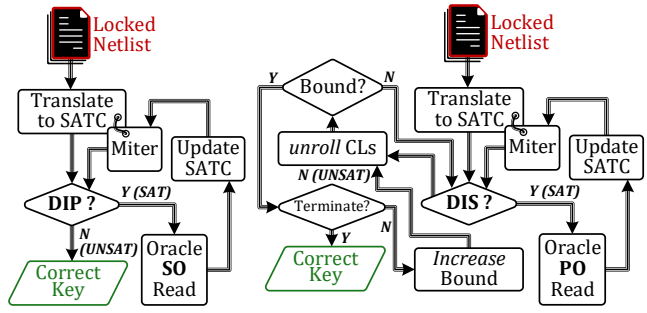
(a) *OPEN* Scan Chain (b) *BLOCKED* Scan Chain

Figure 2: Scan Availability in De-obfuscation Attacks.

Over the last decade, numerous studies have evaluated and challenged the validity and robustness of existing logic locking techniques against different forms of attacks, especially attacks on key-based logic locking techniques [25]. Depending on the threat models defined for the attacks, these attacks could be categorized into different groups. One central assumption in these threat models is the availability of the DFT infrastructure (i.e., scan chain pins) for the adversaries [13]. When the access to the scan infrastructure is *OPEN*, as shown in Fig. 2(a), the adversary would be able to have separate access to inputs/outputs of each combinational logic (CL), separately. By controlling the scan enable (SE), this access could be achieved through the scan in (SI) and the scan out (SO). Assuming that the access to the scan pins is *OPEN*, Boolean satisfiability (SAT)-based attack could break most forms of the logic locking techniques in a matter of minutes [31, 40].

In the SAT attack, as illustrated in Fig. 3(a), the adversary first transforms each CL of the reverse-engineered circuit to SAT circuit (SATC). Then, by building the miter circuit ($CL(pi, K_1) \oplus CL(pi, K_2)$), the adversary revokes the SAT solver to find *pi* which distinguish between two keys (different outputs for K_1, K_2), called *discriminating input pattern* (DIP). Then, this DIP will be queried on the oracle (through DFT), and the SI/SO constraint for the CL will be stored back in the SAT solver, and the miter circuit would be solved again in the next iteration. When the miter+constraints problem no longer has a satisfying assignment in one iteration, the list of added constraints is a complete set that uniquely characterizes a correct key. Finding a correct key is then straightforward. Any key that satisfies this set of constraints is correct, and it could be found using a single query to the SAT solver. With access to the scan pins and using this flow, the adversary would target each CL separately and apply the combinational de-obfuscation attack.

Since the validity and strength of most logic locking techniques are undermined by combinational SAT attacks, numerous studies



(a) Combinational SAT Attack (b) Sequential SAT Attack

Figure 3: Combinational SAT vs. Sequential SAT Attack.

show how the scan access could be limited to avoid the feasibility of such attacks [13, 14, 23, 38, 39, 51]. When the scan chain access is *BLOCKED*, as demonstrated in Fig. 2(b), the adversary's access is limited to PI/PO, and she cannot build the SATC for each CL. However, even while the access to the circuit is limited to PI/PO, the SAT solver could be used to model an attack on the sequential circuits as a whole. The attack procedure on sequential circuits with no scan access is shown in Fig. 3(b). Similar to the SAT attack, it has an iterative structure for pruning the search space. However, due to the restricted access to the internal registers, to formulate the internal state, unfolding could be merged with the combinational SAT attack to extend it for sequential circuits [22, 28, 32]. Hence, rather than finding a DIP in each iteration, it finds a sequence of inputs denoted as *discriminating input sequence* (DIS) that can generate two different outputs for two different keys. After finding each DIS, the miter+constraint will be updated with a new constraint to ensure that the next onset of keys produces the same output for previously found DIS. This process continues until no further DIS is found within the given boundary.

Both combinational and sequential SAT-based de-obfuscation attacks could be applied to key-based logic locking techniques, and they are not directly applicable to key-less logic locking. The key-less logic locking was first introduced in HARPOON [44]. HARPOON is a sequential logic locking technique that modifies the FSM of a circuit. In such key-less techniques, one or few extra sets (modes) of state, such as obfuscation mode or authentication mode as shown in Fig. 1(b), is merged with the original FSM of the circuit. In such an FSM locking solution, a specific *unlocking* sequence is required (and applied in multiple cycles) to drive the FSM from a locked state to reach the active FSM's original initial state [7, 44]. The target of an adversarial attack against such FSM locking solutions is to find a sequence of input patterns in the result of which the initial state of the original FSM is reached and IP is activated.

3 PROPOSED ATTACK FRAMEWORK: RANE

The existing attack frameworks on logic locking with available source codes are developed by exploiting pre-compiled Binary and modulo theory solvers that accept the netlists in Bench, which is a minimal language for the description of hardware [22, 24, 40]. This requirement introduces a burden for modeling and assessing logic locking as all complex structures have to be re-synthesized and expressed in the simplest logic structures compatible by the solver. For instance, a 2-bit Full Adder (FA) in acceptable Bench format is depicted in Fig. 4 and its corresponding Verilog format demonstrated in Fig. 5. As illustrated, the FAs, although available in the standard cell library, cannot be interpreted by the solver's native macros and have to be translated to basic logic gates/macros. Problems become more complicated when complex standard syntax declarations such as vectors, inouts, and aliasing are used. More precisely, the limitations faced during the modeling of a complex netlist in simplified Bench format include (1) limited availability of available macros with inherent support only for the description of basic gates, (2) static syntax declaration for available macros with no possibility of extension, (3) requirement for having/writing a dedicated parser for such format that is library- and language-dependent, (4) incompatibility with many standard syntax declarations, like vector, inout, aliasing, etc. The complexity involved in building a translator and having to model and account for these complexities significantly raises the bar for the application of the existing attacks.

```
INPUT(a[0])
INPUT(a[1])
INPUT(b[0])
INPUT(b[1])
INPUT(cin)

OUTPUT(s[0])
OUTPUT(s[1])

OUTPUT(cout)
```

```
s[0] = XOR(a[0], b[0], cin)
cout_01 = AND(a[0], b[0])
cout_02 = AND(a[0], cin)
cout_03 = AND(b[0], cin)
cout_0 = OR(cout_01, cout_02, cout_03)
s[1] = XOR(a[1], b[1], cout_0)
cout_11 = AND(a[1], b[1])
cout_12 = AND(a[1], cout_0)
cout_13 = AND(b[1], cout_0)
cout = OR(cout_11, cout_12, cout_13)
```

Figure 4: Acceptable BENCH format in Existing and Available SAT and Sequential SAT Attacks' Source Codes [22, 40] for a 2-bit FA.

```
`timescale 1ns / 1ps
//////////////////////
// Lib.v
//////////////////////
module FA_1bit (
  input a,
  input b,
  input cin,
  output s,
  output cout );

  assign {cout,s} = a + b
                  + cin;
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////
// Top.v
//////////////////////
module FA_2bit(
    input [1:0] a, b,
    input cin,
    output [1:0] sum,
    output carry );
  FA_1bit s0(a[0], b[0], cin,
             sum[0], cr0 ) );
  FA_1bit s1(a[1], b[1], cr0,
             sum[1], carry ) );
endmodule
```

Figure 5: Standard Verilog format Acceptable by RANE for a 2-bit FA.

To overcome this shortcoming, we propose RANE as a CAD-based toolbox for evaluating the security of logic locking that applies to a much broader set of applications and circuits. By exploiting open-source toolkits for design analysis and code generation of RTL designs written in standard HDLs, RANE supports parsing and analyzing circuits written, elaborated, or synthesized in standard HDL, such as Verilog. This also allows us to use formal verification tools for the de-obfuscation modeling instead of using pre-compiled solvers as the core of de-obfuscation. The usage of formal verification tools allows RANE to be extended based on the inherent features of these formal tools. Besides, the RANE applicability is seamlessly improved as formal tools are revised and upgraded to parse and interact with new libraries and complex macros without having to do any additional translation or modeling.

3.1 RANE Framework

Fig. 6(a) shows the overview of the RANE framework. In the RANE framework, we provide two different solutions: (1) a formal-based interface through *Pyverilog* generator, and (2) a pre-compiled static-model tool using *PySMT* generator. In the first solution, we use Pyverilog [49] as the open-source HDL analyzer for code parsing, static analysis, and code translation. Pyverilog framework is captured in Fig. 6(b). The parser, dataflow analyzer, control-flow analyzer, and Verilog code generator are the four major features in Pyverilog. Pyverilog also provides a dataflow and control-flow graph visualizer for interpreting the hardware. In RANE, we implement and integrate different interfaces to support different verification and solver tools. As demonstrated in Fig. 6(a), by getting the benefit of Pyverilog, Cadence JasperGold and SymbiYosys are integrated as the formal tools. Also, using Pyverilog, any model like miter circuit, equivalency check, etc., could be generated using behavioral Verilog code, making the model generation for de-obfuscation much easier. RANE also supports features like exporting/importing constraints, automated cycle pre-processing, and Verilog-based attack model generation.

For the second solution, we implement and integrate an interface for embedding PySMT into the RANE framework. PySMT is a solver-agnostic library for fast prototyping of satisfiability

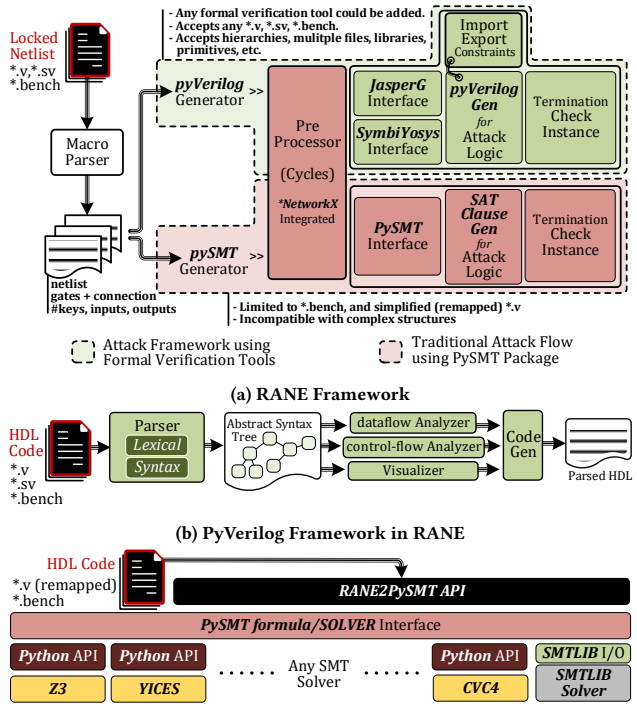


Figure 6: RANE Overall Framework.

modulo theory (SMT)-based algorithms. As demonstrated in Fig. 6(c), by using different APIs, PySMT provides the possibility of invoking well-known SMT solvers, such as Z3 [30], Yices[29], and Boolector[43]. By integrating the PySMT framework, similar to the existing de-obfuscation attack tools, it could be engaged on `Bench` and remapped `Verilog` files over pre-compiled solvers.

3.2 RANE Application

Using this framework, RANE can model different threat models on logic locking and formulate various attacks with much less effort than the existing de-obfuscation attack tools. In the following section, we will evaluate the application of RANE on *FOUR* different case studies: (1) **oracle-less attack on HARPOON**, in which HARPOON is the key-less FSM logic locking, (2) **oracle-guided attack on HARPOON**, (3) **oracle-guided attack on sequential logic locking**, in which random-based logic locking is engaged, and as an assumption of the threat model, the scan chain accessibility is *BLOCKED*, and (4) **oracle-guided attack on combinational logic locking**, in which random-based logic locking is engaged, and the scan chain accessibility is *OPEN*.

3.2.1 ***Case Study 1: Oracle-less Attack on HARPOON.*** In this case study, we assume that the adversary might have only a single working copy of the chip. They can first apply a sequence of input patterns, and by observing the outputs, they can build a database of such I/O pairs. Alternatively, the I/O pair also could be obtained by the adversary at the foundry from the pre-generated functional test patterns or post-layout verification test[1]. Then, by reverse-engineering the chip or having access to the layout at the foundry,

[1]Since HARPOON is a key-less logic locking, functional test patterns or post-layout verification tests could be used to build and extend the database of I/O pair.

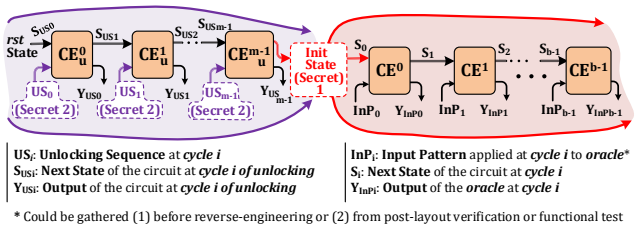


Figure 7: Oracle-Less Attack Model on HARPOON.

Algorithm 1 Oracle-less Attack Model on HARPOON using RANE

```
——————— Formulating of Secret 1 (init state) ———————
Get an I/O sequence ( {InP_0, Y_InP0}, ..., {InP_b-1, Y_InPb-1}) from C_BlackBox;
Model ← CE(InP_0, Ŝ_init, Y_InP0, Ŝ_1) ∧_{i=1}^{|I/O|} CE(InP_i, Ŝ_i, Y_InPi, Ŝ_i+1);
——— Formulating of Secret 2 (unlocking sequence) ———
Model ∧ = CE^0_u(US_0, S_rst, Y_US0, S_US1);
i ← 1;
——— Invoking the Formal Tool: Finding Secret 1, 2 ———
while Formal(Model ∧ (S_USi = Ŝ_init)) → Fail do
    Model ∧ = CE^i_u(US_i, S_USi, Y_i, S_USi+1);
    i ← i + 1;
end while
return Formal(Model ∧ (S_USi = Ŝ_init))   ▷ {init state, unlocking sequence}
```

the netlist could be extracted. We refer to this attack model as *bronze model*. The adversary in this model does not have access to the scan chain. By using this threat model, the de-obfuscation attack on HARPOON could be accomplished in two main steps: (1) Finding the initial value of FFs (*init state*), e.g., S_0 in Fig. 1(b), such that if the init state initializes the circuit, it would produce the same output if the input patterns are applied to the oracle; (2) Formulating the formal verification problem to find the correct sequence of input patterns, allowing us to reach the previously found init state, referred to as *unlocking sequence*. For example, in Fig. 1(b), $pi_1 \rightarrow pi_7$ is the unlocking sequence to reach the init state.

Fig. 7 shows the oracle-less attack model on HARPOON based on the bronze threat model. For the first step of the attack, i.e., formulating *secret 1* of Fig. 7, the init state is considered as the key. Then, by applying the I/O sequences from the pre-built database to the (unrolled) combinational equivalent (CE) netlist[2], i.e. $InP_0 \rightarrow (CE^0)$, $InP_1 \rightarrow (CE^1)$, etc., constraining that the output values (Y_{InP0}, Y_{InP1}, etc.) must match with oracle outputs, the init state could be found by formal tool. After formulating the *secret 1*, then the second step will be formulated for finding the unlocking sequence, i.e. finding *secret 2* of Fig. 7. In this step, unlocking sequence is the key, i.e., $US_{0:m-1}$, and with constraining that state of the circuit that must reach the valid init state, the formal tool integrated with the RANE framework could find the unlocking sequence. Algorithm 1 also illustrates the flow of this case study in the RANE framework. It consists of three steps: (1) formulating of *secret 1*, (2) formulating of *secret 2*, and (3) invoking the formal tool for finding both secrets. Note that all unrolling steps are implicitly done by the formal tool.

It is worth mentioning that if the adversary aims to *ONLY* reverse engineer the chip, formulating and performing part 1 and part 3 of the Algorithm 1 would be enough. After finding the init state, the adversary could then insert their own scan chain into the reverse-engineered netlist to provide the possibility of loading the proper init state to the FFs, bypassing the need to go through the second step for finding the unlocking sequence. Furthermore, assuming

[2]all CE^is/CE^i_us are the same each represents the combinational equivalent of the locked netlist. Each CE^is/CE^i_us is implicitly generated for one cycle by the formal tool.

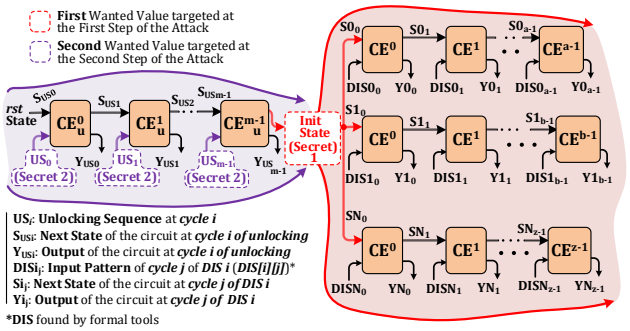


Figure 8: Oracle-Guided Attack Model on HARPOON.

Algorithm 2 Oracle-guided Attack Model on HARPOON using RANE

```
------- Finding Secret 1 (init state) -------
Model ← C_seq(X, S_init1, Y_1) ∧ C_seq(X, S_init2, Y_2);
while !UC(Model) ∧ !CE(Model) ∧ !UMC(Model) do
    DIS_i ← Formal(Model ∧ (Y_1 ≠ Y_2));
    Y_i ← C_BlackBox(DIS_i);
    Model ∧= C_seq(DIS_i, S_init1, Y_i) ∧ C_seq(DIS_i, S_init2, Y_i);
end while
------- Finding Secret 2 (unlocking sequence) -------
Model ∧= CE^0_u(US_0, S_rst, Y_US0, S_US1);
i ← 1;
while Formal(Model ∧ (S_USi = Ŝ_init)) → Fail do
    Model ∧= CE^i_u(US_i, S_USi, Y_i, S_USi+1);
    i ← i + 1;
end while
return Formal(Model ∧ (S_USi = Ŝ_init))   ▷ {init state, unlocking sequence}
```

that for the pre-built I/O pairs, there exists multiple init states satisfying the formal mode, and each init state could be reached via a unique unlocking sequence, formulating and solving both steps together will automatically constrain finding the valid init state whose unlocking sequence is the shortest one. This case study in RANE is the first of its kind in de-obfuscation attacks that target key-less logic locking techniques like HARPOON with no need for oracle. In our experimental results, we show how the success rate of this model depends on the length/size of I/O pairs available in the pre-built database and the size of the unlocking sequence (numbers of obfuscation/authentication FSMs). Note that since the adversary's capability is limited to only using the available and pre-built sequence of I/O pairs, the existing combinational/sequential SAT can not be used for this attack. This is because the solver can no longer constrain the input patterns freely. But this attack could be easily modeled and carried by RANE.

3.2.2 ***Case Study 2: Oracle-guided Attack on HARPOON.*** In this case study, we target the same logic locking technique evaluated in case study 1, i.e., HARPOON. We assume that the adversary has access to the reverse-engineered netlist and the functional chip (oracle). Other assumptions are the same as the threat model of case study 1. Fig. 8 illustrates this attack model on HARPOON. Similar to case study 1, it could be done in two steps, i.e., finding init state (secret 1) and finding the unlocking sequence (secret 2). However, these two steps will be accomplished in sequence. Regarding the first step (secret 1), since the oracle is available for the adversary, the generation of the sequence for finding the init state will be done by the formal tool. Similarly, all unrolling operations will be done implicitly in this case study. The availability of the oracle also allows us to expand the number of sequences from one to many ($InP1_{1:a}$, $InP2_{1:b}$, ..., $InPN_{1:z}$). Unlike the attack model in case study 1, since the adversary's capability is not limited to a fixed I/O pair database, this model's success rate does not depend on the length/size of the sequences. Algorithm 2 depicts the flow of case study 2. Note that unlike case study 1 that finds both secrets at once, in this case, multiple formal tool invocation will be accomplished for finding secret 1 followed by finding secret 2.

Algorithm 3 Oracle-guided Attack Model on Logic Locking with *BLOCKED* scan chain (Sequential Logic Locking) using RANE

```
Model = C_seq(X, K_1, Y_1) ∧ C_seq(X, K_2, Y_2);
while !UC(Model) ∧ !CE(Model) ∧ !UMC(Model) do
    X_dis ← Formal(Model ∧ (Y_1 ≠ Y_2));
    Y_f ← C_BlackBox(X_dis);
    Model ∧= C_seq(X_dis, K_1, Y_f) ∧ C_seq(X_dis, K_2, Y_f);
end while
return Formal(Model)   ▷ Return Correct Key
```

Since formal tool is employed for finding the DISes, the termination condition would be adopted from conventional sequential SAT attack [32]: (1) **Unique Completion (UC):** This criterion checks for the uniqueness of the secret. If there is only a single secret that satisfying the defined constraints, the attack is terminated. (2) **Combinational Equivalence (CE):** If there is more than one secret that agrees with the constraints, the attack checks the combinational equivalency of the remaining secrets. In this step, the input/output of FFs is considered as pseudo primary outputs/inputs allowing the attacker to treat the circuit as combinational. The resulting circuit is subjected to an SAT attack. If the SAT solver fails to find a different output or next state for two different secrets, it concludes that all remaining secrets are correct, and the attack terminates. (3) **Unbounded Model Check (UMC):** If UC and CE fail, the attack checks the existence of a satisfying assignment for the remaining secrets using an unbounded model checker. This is an exhaustive search with no limitation on bound.

3.2.3 ***Case Study 3: Oracle-guided Attack on Sequential Logic Locking.*** RANE attack could also be used for breaking conventional key-based logic locking solutions, such as random logic locking (RLL) [18], or strong logic locking (SLL) [17] applied on the sequential circuit, where access to the scan chain is *BLOCKED*. In this case study, we target to model the conventional SAT-based sequential de-obfuscation attack, shown in Fig. 3(b). The adversary has access to the PI/PO of the oracle and reverse-engineered *locked* netlist. Unlike the existing sequential de-obfuscation attacks [22, 32] that handle the unrolling explicitly by the framework, RANE could accomplish it both implicitly handled by the formal verification tool encapsulated in the RANE framework or explicitly by the defined attack model. Also, supporting `Verilog` in the RANE framework allows us to get the benefit of behavioral `Verilog` helping to build any model with much less effort.

The support of implicit unrolling provides the RANE framework to use any of the available either open-source or commercial verification tools. Hence, RANE can get the benefit of the scalability, stability, and adaptability of these tools to handle a much richer set of input formats, handle a wider range of gates[3]. This is the main aim of the RANE framework that be easily adaptable in any flow,

[3]Formal tools could support any type of macros defined in the standard cell library, as opposed to very limited basic gates available in Bench format (used in the existing sequential de-obfuscation attack, i.e., KC2).

Algorithm 4 Oracle-guided Attack Model on Logic Locking with *OPEN* scan chain (Combinational Logic Locking) using RANE

1: $Model = C_{comb\_lock}(X, K_1, Y_1) \wedge C_{comb\_lock}(X, K_2, Y_2)$;
2: **while** $Formal(Model \wedge (Y_1 \neq Y_2))$ **do**
3: $X_{dip} \leftarrow Formal(Model)$;
4: $Y_f \leftarrow C_{BlackBox}(X_{dip})$;
5: $Model \wedge = C_{comb\_lock}(X_{dip}, K_1, Y_f) \wedge C_{comb\_lock}(X_{dip}, K_2, Y_f)$;
6: **end while**
7: **return** $Formal(Model)$ ▷ Return Correct Key

without the need for input format translation, remapping, Decoding, re-synthesis. Algorithm 3 shows the flow of case study 3 in the RANE framework with implicit unrolling. As demonstrated, the attack formulation is first initiated using the miter circuit (XORed double-circuit). Then, per each iteration, the formal tool looks for a DIS and two keys that produce different outputs for that DIS. In the next iterations, the previously found DISes must match with the oracle, and the attack model termination conditions will be checked when no more DIS is found. As shown, unrolling operations for finding DISes are not formulated in the model (implicit unrolling), and it will be handled automatically by the formal tool.

3.2.4 ***Case Study 4: Oracle-guided Attack on Combinational Logic Locking.*** In this case study, RANE emulates the most well known SAT-based attack on logic locking proposed by Subramanyan *et al.* [40], which is oracle-guided on logic lockings with *OPEN* scan chain access, referred to as SAT-based combinational de-obfuscation attack. As demonstrated in Algorithm 4, in the SAT-based combinational de-obfuscation attack, a (distinguishing) miter circuit needs to be built as $miter \equiv C_{comb\_lock}(X, K_1) \neq C_{comb\_lock}(X, K_2)$ for any arbitrary locked combinational logic C_{comb_lock}. Based on the miter circuit, the formal tool will be invoked and will return a DIP that produces different outputs for two different keys. Then, this DIP is queried on the oracle, $C_{BlackBox}$, $eval \leftarrow C_{BlackBox}(X_{dip})$ and the I/O-constraint for the equivalency check, $C_{comb\_lock}(X_{dip}, K_1) = C_{comb\_lock}(X_{dip}, K_2) = eval$ will be added as a new constraint to the formal tool, and after this update, the $miter$ circuit would be solved again. When the $miter + constraints$ problem has no satisfying assignment (no more DIP), it could identify the correct key.

From the formal tool perspective in RANE, the formulation of both key-based oracle-guided combinational and sequential de-obfuscation attacks are very similar. The only difference is that for the attack model on the sequential circuits, the formal tool looks for DIS (with implicit unrolling), but in the model on the combinational circuit, finding DIP is the main objective of the formal tool.

4 EXPERIMENTAL RESULTS

With exploiting packages like PySMT [33] and Pyverilog [49], the proposed RANE framework has been implemented in `Python3`. The current version of the RANE framework, available in [8], has been built over different formal verification tools configurable by the users, including Cadence JasperGold as the commercial formal verification tool and SymbiYosys as a formal open-source tool. The formal tools are responsible for major operations of attack modeling, such as unrolling, building miter, finding sequences, DIPs, and DISes. In this paper, the experiments are accomplished using the open-source SymbiYosys formal verification engine[4]. We evaluate and verify the feasibility/performance of the RANE framework, based on all *FOUR* case studies previously discussed in Section 3.2, on a set of ISCAS-{85/89} benchmark circuits, as listed in Table 1. For sequential-based experiments, i.e., case studies 1, 2, and 3, since the circuits have a sequential depth of fewer than 100 cycles, with skipping UMC check, the boundary/depth is set to 100 cycles. The integration of PySMT and SymbiYosys allows RANE to get the benefit of different solvers. In the experiments, and based on our observation to get the most benefit, we use *Yices* for case studies 1 and 2, and the best performance achieved by *Boolector, MathSAT,* or *Yices* for case studies 3 and 4. All experiments are carried on ARGO cluster computing [9] as a computing cluster equipped with Intel Xeon E5-2670, with 16 core CPUs and 512GB of RAM.

Table 2 demonstrates the performance of the RANE framework when it is configured for case study 1, in which the circuits are locked with HARPOON [44]. When the circuit's state is in obfuscation/authentication modes, random POs are selected to be corrupted. We also used random input patterns to build the database of I/O pairs for this case study. In this experiment, the number of authentication/obfuscation FSMs (unlocking sequence size) is swept. As shown, for different circuits, with a different number of obfuscation FSMs (different unlocking sequence sizes), the threat model defined in case study 1 can retrieve the secrets with a small number of I/Os.

Since the output of this model is based on a limited set of pre-built I/O pairs, this threat model cannot guarantee the uniqueness of the init state (secret 1) generated by the framework. However, increasing the size of I/O pairs, or applying different sets of random I/O pairs, results in restricting the different {secret 1, secret 2} possibilities that match with the oracle pre-generated I/O pairs. In this experiment, we limit the size of the pre-built I/O database to 3,000 cycles and by using this size, our observation shows that, on average, for 73.4% of the cases in Table 2, the extracted secrets are the correct expected ones. Note that, since this model generates the attack model once (without iterative structure)[5], increasing the number of I/O pairs does not affect the execution time significantly (almost linearly *w.r.t.* the number of I/O pairs).

Table 3 depicts the performance of the RANE framework on the same logic locking technique, i.e., HARPOON[6], but in this case (case study 2), we assumed that the oracle is available. In this case, the formal tool can generate different DISes for finding init state. Unlike case study 1, finding the unlocking sequence (secret 2) will be started when the init state (secret 1) is found. Hence, in this

Table 1: Description of ISCAS-85/89* circuits.

Circuit	#Gates	#PIs	#POs	Circuit	#Gates	#PIs	#POs	Circuit	#Gates	#PIs	#POs
c432	160	36	7	c1355	546	41	32	c3540	1,669	50	22
c499	202	41	32	c1908	880	33	25	c5315	2,307	178	123
c880	383	60	26	c2670	1,269	233	140	c7552	3,513	207	108
Circuit	#FFs	#PIs	#POs	Circuit	#FFs	#PIs	#POs	Circuit	#FFs	#PIs	#POs
s344	15	9	11	s832	5	18	19	s5378	179	35	49
s382	21	3	6	s838	32	34	1	s13207	638	62	152
s386	6	7	7	s1196	18	14	14	s15850	534	77	150
s526	21	3	6	s1423	74	17	5	s35932	1,728	35	320
s713	19	35	23	s1494	6	8	19	s38584	1,426	38	304

*s9234 *MUST* be ignored since it has some FFs that have no path to POs.

[4]To facilitate re-producing the results by the community and remove the dependency on commercial tools, the results are generated on available open-source tools. Our preliminary investigation shows that the results could improve significantly (in terms of runtime and memory) when a commercial tool, such as Cadence JasperGold, is configured as the utilized formal method tool.

[5]The model defined in Fig. 7 will be generated at once. The attack process will be accomplished for two secrets simultaneously. It will find the init state (secret 1) and the unlocking sequence (secret 2) at once.

[6]The same locked circuits are used for the experiments on case studies 1 and 2.

Table 2: RANE performance in Case Study 1 - Oracle-less on HARPOON.

Circuit	{3, 18}*		{5, 30}		{10, 60}		{20, 120}	
	time	#I/O	time	#I/O	time	#I/O	time	#I/O
s344	4	20	4	30	5	60	22	120
s382	1	20	-	-	4	60	172	120
s386	4	20	5	30	7	60	22	120
s526	-+	-	-	-	-	-	-	-
s713	5	20	6	30	8	60	23	120
s832	5	20	5	30	-	-	-	-
s838	-	-	3	30	6	60	22	120
s1169	6	50	20	230	8	60	-	-
s1423	4	30	10	110	11	100	65	120
s1494	224	20	226	30	249	60	1,468	120
s5378	6	20	7	30	15	60	76	120
s13207	15	20	22	30	56	60	246	120
s15850	-	-	18	30	40	60	-	-
s35932	-	-	53	30	111	60	-	-
s38417	72	50	-	-	-	-	-	-
s38584	49	20	108	60	-	-	1,117	120

*{number of obfuscation/authentication FSMs, The length of unlocking sequence}
time: in Seconds #I/O: number of input/output patterns
+Failed to find the correct init state by using 3,000 I/O pairs.

experiment, the execution time of the RANE framework is divided into two parts, $t_1 + t_2$, in which t_1 is the RANE execution time for finding the secret 1, and t_2 is the time required to find secret 2. The *size* indicates the attack model size in terms of {*#DISes/Depth*}. This experiment reveals one of the biggest limitations of unrolling-based attacks. The problem size will be grown in two dimensions: (1) increasing the number of DISes, (2) increasing the depth of unrolling. Thus, for larger circuits, this model faces a larger execution time. For cases with the memory bound, switching to commercial formal tools, e.g. Cadence JasperGold, will resolve the issue.

Table 4 shows the performance of the RANE framework in case study 3. In this case, we assume the (XOR-based) key gates are inserted at random places, the access to the scan chain is *BLOCKED*, and the attack model evaluates the circuit as a whole. To provide comparative results, we engage the PySMT generator for building the model for this case study. The unrolling has been accomplished statically/explicitly, and similar to KC2 (neos), the locked circuits are in Bench format. In this experiment, the best performance achieved by *Boolector, MathSAT,* and *Yices* has been reported [43]. As demonstrated, with outperforming KC2 (neos) [22] for larger circuits, the RANE framework promises better scalability. Note that, since KC2 (neos) is a pre-compiled C++ platform, it outperforms RANE for the smaller circuits[7].

Table 3: RANE performance in Case Study 2 - Oracle-guided on HARPOON.

Circuit	{3, 18}*		{5, 30}		{10, 60}		{20, 120}	
	time	size	time	size	time	size	time	size
s344	3+1	1/2	5+2	3/4	11+9	4/4	20+98	7/4
s382	8,484+107	107/44	8,101+218	93/44	4,449+398	65/67	to	to
s386	3+1	3/3	12+3	11/4	30+9	12/4	100+195	15/4
s526	1,644+28	59/44	17,749+275	129/44	2,677+275	48/50	6,4441+to	48/49
s713	102+3	7/8	144+6	9/10	496+17	9/10	1,322+619	43/6
s832	10+2	7/10	6+2	4/5	59+23	17/11	34+113	12/13
s838	3,175+82	45/66	293+29	43/4	701+140	63/6	4,319+3,742	112/18
s1196	67+10	13/15	45+4	10/11	79+42	14/15	79+148	11/13
s1423	26,300+556	157/16	to	-	to	-	to	-
s1494	11+3	9/6	12+5	7/8	41+36	13/18	26+99	9/5
s5378	323+8	26/9	532+67	34/10	722+148	32/9	1,840+829	34/9
s13207	68,817+2,839	74/29	77,102+3,772	73/29	to	-	mem	-
s15850	to	-	to	-	to	-	to	-
s35932	1,167+1,189	21/7	1,234+1,815	22/7	mem	-	mem	-
s38417	mem	-	mem	-	mem	-	mem	-
s38584	mem	-	mem	-	mem	-	mem	-

*{number of obfuscation/authentication FSMs, The length of unlocking sequence}
time: in Seconds timeout (to): 24 hours size: #DISes/Depth
mem: Out of memory

[7] compared to RANE, KC2 provides faster parsing/solving on the small circuits.

Table 4: RANE performance comparison with KC2 [22] in Case Study 3 - Oracle-guided on Random-based Sequential Logic Locking.

Circuit	RANE					KC2 (neos) [22]				
key size	100	150	200	250	300	100	150	200	250	300
s344	5	7	n/a	n/a	n/a	1	5	n/a	n/a	n/a
s386	6	30	n/a	n/a	n/a	3	27	n/a	n/a	n/a
s832	11	40	196	504	n/a	3	50	644	5,771	n/a
s1196	4	14	9	15	70	1	5	4	10	82
s1494	20	31	69	310	511	3	10	18	63	144
key size	10	20	30	40	50	10	20	30	40	50
s382	6	183	to	to	to	68	246	to	to	to
s526	15	4,932	to	to	to	to	to	to	to	to
s713	2	2	1	2	3	0	0	0	0	1
s838	12	211	to	16	to	53	to	to	to	to
s1423	7	21	921	to	to	10	77	9,041	to	to
s5378	16	14	62	25	28	8	10	40	7	9
s13207	779	817	784	820	839	2,840	3,686	2,881	2,790	2,414
s15850	772	738	767	735	921	688	490	507	703	795
s35932	895	660	985	850	811	777	882	3,576	1,095	8,056
s38417	5,571	6,036	6,287	to	to	to	to	to	to	to
s38584	to	to	to	to	to	to	to	to	to	to

time: in Seconds timeout (to): 4 hrs n/a: Circuits are too small for that key size
KC2 is executed with different configurations, and the best performance is reported.

Table 5: RANE performance comparison with SAT attack [40] in Case Study 4 - Oracle-guided on Combinational Logic Locking.

Circuit	RANE				SAT (sld) [40]			
overhead	%5	%10	%25	%50	%5	%10	%25	%50
c432	1	1	1	1	0	0	0	0
c499	0	1	1	2	0	2	2	12
c880	0	1	2	7	0	0	1	4
c1355	1	2	8	107	0	1	7	169
c1908	1	2	20	689	1	2	21	377
c2670	1	to	to	to	to	to	to	to
c3540	2	4	9	201	4	2	6	122
c5315	7	45	to	9,804	5	20	to	to
c7552	44	2,227	to	to	43	to	to	to

time: in Seconds timeout (to): 4 hrs

Table 5 compares the performance of the RANE framework, once it models case study 4 using PySMT generator, with the conventional SAT attack on combinational logic locking by Subramanyan *et al.* [40]. Similarly, since the conventional SAT attack has been deployed using a compiled binary file and uses a pre-compiled SAT solver, it outperforms the RANE framework in some parts of the experiment. However, for larger circuits, we observe that the RANE framework can outperform the conventional SAT attack by reconfiguring it to use the most suitable SAT solver for each circuit.

5 LOOKING INTO FUTURE

RANE integrates CAD formal verification tools with packages like Pyverilog and PySMT to provide rich extensibility, deployability, scalability, and performance in the RANE framework, especially compared to the existing pre-compiled and optimized but static de-obfuscation attacks:

(1) Almost all attacks with much more capabilities and assumptions can be modeled using the RANE framework. For instance, the uncommon usage of latches for latch-based or clock-gated logic locking techniques [19] requires a troublesome and hard-to-be-achieved transition to be acceptable by the existing attacks. However, RANE can support a much more comprehensive range of digital building blocks and macros with full support on standard library cells.

(2) The futuristic support of different features/capabilities in formal tools could be easily engaged in the RANE framework. For instance, the support of range equivalent circuits in the formal tools will allow us to formulate a much more scalable attack model on sequential logic locking. Range equivalent circuits are compressed

circuits representing the circuit's τ^{th} unroll, accepting inputs and generating outputs in the range of the τ^{th} unroll circuit. Although the concept of range equivalent circuits is an open research problem, the RANE framework can support and engage such features once the formal tools support it.
(3) Parallelism could be engaged much more appropriately when CAD formal tools are in place. For instance, the joint Cadence and AWS proof-of-concept for utilizing various degrees of parallelism using JasperGold on AWS (JAWS) show how verification could achieve huge speed-up on parallel computing systems [1].

6 CONCLUSION

In this paper, we introduced the Reversal Assessment of Netlist Encryption (RANE) Attack, an open-source framework for evaluating the security of logic locking techniques. The RANE framework integrates packages like Pyverilog and PySMT with formal verification tools to support circuits described in standard languages, like `Verilog`. We evaluated the effectiveness of the RANE framework on *FOUR* different case studies. We illustrated how the RANE attack could model different de-obfuscation attacks with much less effort. We also demonstrated how the RANE attack could use either a golden chip as a reference or a set of pre-recorded I/Os for the unlocking process. Moreover, we also illustrated how the RANE framework could formulate the first attack model on key-less FSM obfuscation solutions. Our experimental results show that the high scalability of formal tools allows RANE to outperform the existing de-obfuscation attacks on larger circuits while eliminating the shortcomings of prior art de-obfuscation solutions for dealing with translation and modeling of complex structures.

REFERENCES

[1] A. Elzeftawi *et al.* 2021. Running Cadence JasperGold formal verification on AWS at scale. https://aws.amazon.com/blogs/industries/tag/cadence-jaspergold/.
[2] A. R. Desai *et al.* 2013. Interlocking Obfuscation for Anti-Tamper Hardware. In *Proc. of the Cyber-Security and Information Research Workshop*. 1–8.
[3] A. Yeh. 2012. Trends in the Global IC Design Service Market. *DIGITIMES* (2012).
[4] B. Shakya *et al.* 2020. CAS-lock: A Security-Corruptibility Trade-off Resilient Logic Locking Scheme. *IACR Trans. on Cryptographic Hardware and Embedded Systems (CHES)* (2020), 175–202.
[5] C. Wolf *et al.* 2013. Yosys-a free Verilog synthesis suite. In *Proc. of Austrian Workshop on Microelectronics (Austrochip)*.
[6] D. Sirone, and P. Subramanayan. 2020. Functional analysis attacks on logic locking. *IEEE TIFS* (2020).
[7] F. Koushanfar. 2017. Active Hardware Metering by Finite State Machine Obfuscation. In *Hardware Protection through Obfuscation*. 161–187.
[8] GATE Lab. 2021. RANE framework v.1.0.0. https://github.com/gate-lab/RANE, https://cadforassurance.org/tools/evaluation-of-obfuscation/rane.
[9] GMU Office of Research Computing. 2021. ARGO Cluster. http://wiki.orc.gmu.edu/mediawiki.
[10] H. M. Kamali *et al.* 2018. LUT-lock: A Novel LUT-based Logic Obfuscation for FPGA-bitstream and ASIC-hardware Protection. In *IEEE Annual Symp. on VLSI (ISVLSI)*. 405–410.
[11] H. M. Kamali *et al.* 2019. Full-Lock: Hard Distributions of SAT Instances for Obfuscating Circuits using Fully Configurable Logic and Routing Blocks. In *Design Automation Conf. (DAC)*. 89.
[12] H. M. Kamali *et al.* 2020. InterLock: An Intercorrelated Logic and Routing Locking. In *Int'l Conf. On Computer Aided Design (ICCAD)*. 1–9.
[13] H. M. Kamali *et al.* 2020. On Designing Secure and Robust Scan Chain for Protecting Obfuscated Logic. In *Great Lakes Symp. on VLSI (GLSVLSI)*. 217–222.
[14] H. M. Kamali *et al.* 2020. SCRAMBLE: The State, Connectivity and Routing Augmentation Model for Building Logic Encryption. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 153–159.
[15] H. Zhou *et al.* 2017. CycSAT: SAT-based Attack on Cyclic Logic Encryptions. In *ICCAD*. 49–56.
[16] J. Dofe *et al.* 2018. Novel Dynamic State-Deflection Method for Gate-Level Design Obfuscation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 37, 2 (2018), 273–285.
[17] J. Rajendran *et al.* 2012. Security Analysis of Logic Obfuscation. In *Design Automation Conf. (DAC)*. 83–89.
[18] J. Roy *et al.* 2008. EPIC: Ending Piracy of Integrated Circuits. In *Design, Automation & Test in Europe Conf. (DATE)*. 1069–1074.
[19] J. Sweeney *et al.* 2020. Latch-Based Logic Locking. In *Int'l Symp. on Hardware Oriented Security and Trust (HOST)*. 132–141.
[20] K. Shamsi *et al.* 2017. AppSAT: Approximately Deobfuscating Integrated Circuits. In *HOST*. 95–100.
[21] K. Shamsi *et al.* 2017. Cyclic Obfuscation for Creating SAT-Unresolvable Circuits. In *GLSVLSI*. 173–178.
[22] K. Shamsi *et al.* 2019. KC2: Key-Condition Crunching for Fast Sequential Circuit Deobfuscation. In *Design, Automation & Test in Europe Conf. (DATE)*. 534–539.
[23] K. Z. Azar *et al.* 2019. COMA: Communication and Obfuscation Management Architecture. In *Int'l Symp. on Research in Attacks, Intrusions and Defenses (RAID)*. 181–195.
[24] K. Z. Azar *et al.* 2019. SMT Attack: Next Generation Attack on Obfuscated Circuits with Capabilities and Performance Beyond the SAT Attacks. *IACR Trans. on Cryptographic Hardware and Embedded Systems (CHES)* (2019), 97–122.
[25] K. Z. Azar *et al.* 2019. Threats on Logic Locking: A Decade Later. In *Great Lakes Symp. on VLSI (GLSVLSI)*. 471–476.
[26] K. Z. Azar *et al.* 2021. Data Flow Obfuscation: A New Paradigm for Obfuscating Circuits. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 29, 4 (2021).
[27] K. Z. Azar, H. M. Kamali, H. Homayoun, A. Sasan. 2020. NNgSAT: Neural Network guided SAT Attackon Logic Locked Complex Structures. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–9.
[28] L. Alrahis *et al.* 2019. ScanSAT: Unlocking Obfuscated Scan Chains. In *ASP-DAC*.
[29] L. De Moura. 2014. Yices 2.2. In *Int'l Conf. on Computer Aided Verification*.
[30] L. De Moura *et al.* 2008. Z3: An Efficient SMT Solver. In *Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.
[31] M. El Massad *et al.* 2015. Integrated Circuit (IC) Decamouflaging: Reverse Engineering Camouflaged ICs within Minutes. In *NDSS*.
[32] M. El Massad *et al.* 2017. Reverse Engineering Camouflaged Sequential Circuits without Scan Access. In *Int'l Conf. On Computer Aided Design (ICCAD)*. 33–40.
[33] M. Gario *et al.* . 2015. PySMT: A Solver-agnostic Library for Fast Prototyping of SMT-based Algorithms. In *SMT workshop*.
[34] M. Rostami *et al.* 2014. A Primer on Hardware Security: Models, Methods, and Metrics. *Proc. of the IEEE* 102, 8 (2014), 1283–1295.
[35] M. Yasin *et al.* 2016. SARLock: SAT Attack Resistant Logic Locking. In *HOST*.
[36] M. Yasin *et al.* 2017. Provably-Secure Logic Locking: From Theory to Practice. In *ACM SIGSAC Conf. on Computer and Communications Security (CCS)*. 1601–1618.
[37] N. Limaye and O. Sinanoglu. 2020. DynUnlock: Unlocking Scan Chains Obfuscated using Dynamic Keys. In *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*. 270–273.
[38] N. Limaye *et al.* . 2020. Thwarting All Logic Locking Attacks: Dishonest Oracle with Truly Random Logic Locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020).
[39] N. Limaye *et al.* 2019. Is Robust Design-for-Security Robust Enough? Attack on Locked Circuits with Restricted Scan. In *ICCAD*. 1–8.
[40] P. Subramanyan *et al.* 2015. Evaluating the Security of Logic Encryption Algorithms. In *Int'l Symp. on Hardware Oriented Security and Trust (HOST)*. 137–143.
[41] P. Tuyls *et al.* 2006. Read-Proof Hardware from Protective Coatings. In *Int'l Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. 369–383.
[42] R. Brayton *et al.* 2010. ABC: An Academic Industrial-strength Verification Tool. In *Int'l Conf. on Computer Aided Verification*. 24–40.
[43] R. Brummayer *et al.* 2009. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. 174–177.
[44] R. S. Chakraborty *et al.* 2009. HARPOON: An Obfuscation-based SoC Design Methodology for Hardware Protection. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 28, 10 (2009), 1493–1502.
[45] S. Roshanisefat *et al.* 2018. Benchmarking the Capabilities and Limitations of SAT Solvers in Defeating Obfuscation Schemes. In *IEEE Int'l Symp. on On-Line Testing And Robust System Design (IOLTS)*. 275–280.
[46] S. Roshanisefat *et al.* 2018. SRCLock: SAT-resistant cyclic logic locking for protecting the hardware. In *Great Lakes Symposium on VLSI (GLSVLSI)*. 153–158.
[47] S. Roshanisefat *et al.* 2020. DFSSD: Deep Faults and Shallow State Duality, a Provably Strong Obfuscation Solution for Circuits with Restricted Access to Scan Chain. In *IEEE VLSI Test Symp. (VTS)*. 1–6.
[48] S. Roshanisefat *et al.* 2020. SAT-Hard Cyclic Logic Obfuscation for Protecting the IP in the Manufacturing Supply Chain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 4 (2020), 954–967.
[49] S. Takamaeda-Yamazaki. 2015. Pyverilog: A Python-based Hardware Design Processing Toolkit for Verilog HDL. In *Int'l Symp. on Applied Reconfigurable Computing*. 451–460.
[50] T. Meade *et al.* 2019. Neta: when ip fails, secrets leak. In *Proc. of Asia and South Pacific Design Automation Conf. (ASP-DAC)*. 90–95.
[51] U. Guin *et al.* . 2018. Robust design-for-security architecture for enabling trust in ic manufacturing and test. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 26, 5 (2018), 818–830.
[52] X. Xu *et al.* 2017. Novel Bypass Attack and BDD-based Trade-off against all Known Logic Locking Attacks. In *CHES*. 189–210.
[53] Y. Xie, and A. Srivastava. 2016. Mitigating SAT Attack on Logic Locking. In *CHES*.