DE2: SAT-based Sequential Logic Decryption with a Functional Description

You Li*, Guannan Zhao*, Yunqi He, and Hai Zhou
Northwestern University, Evanston, USA
{you.li, gnzhao, yunqi.he}@u.northwestern.edu, haizhou@northwestern.edu

***Abstract*—Logic locking is a promising approach to protect the intellectual properties of integrated circuits. Existing logic locking schemes assume that an adversary must possess a cycle-accurate oracle circuit to launch an I/O attack. This paper presents DE2, a novel and rigorous attacking algorithm based on a new adversarial model. DE2 only takes a high-level functional specification of the victim chip. Such specifications are increasingly prevalent in the modern IC design flow. DE2 closes the timing gap between the specification and the circuit with an automatic alignment mechanism, which enables effective logic decryption without cycle-accurate information. An essential enabler of DE2 is a synthesis-based sequential logic decryption algorithm called LIM, which introduces only a minimal overhead in every iteration. Experiments show that DE2 can efficiently attack logic-locked benchmarks without access to a cycle-accurate oracle circuit. Besides, LIM can solve 20% more ISCAS'89 benchmarks than state-of-the-art sequential logic decryption algorithms.**

I. Introduction

Logic locking [1] is a prominent approach that can protect the intellectual properties and enhance the security of hardware designs. Figure 1 depicts the general IC design flow with logic locking. The design house introduces key-controlled protection logic into the circuit during logic or physical design. Once the fabricated chip is returned to the design house, they are activated by applying the pre-selected key. Without knowing a correct key, an adversary cannot fully recover the original functionalities of the chip.

Logic decryption aims to reveal a correct key with structural or behavioral analysis. Since proposed in 2015, the SAT attack [2] has established the status quo of logic decryption. It provides solid guarantees on both termination and correctness of the returned key. Although there exist defense mechanisms to thwart this attack [3], [4], they face a trilemma among SAT resilience, structural robustness, and locking efficiency [5]. Therefore, the SAT attack remains a prominent approach to logic decryption.

To launch the SAT attack, an adversary needs to obtain *i)* the logic-locked netlist and *ii)* a working chip as the *oracle circuit*. The same requirement applies to all *oracle-guided* attacks. In reality, an adversary can acquire the logic-locked netlist through a rogue insider within the design house or recover it from a physical layout obtained from a foundry or assembly facility [6]. On the other hand, acquiring a working chip can be challenging in many scenarios. For example, *a)* a chip may be designed for mission-critical applications or is fully customized, so a working chip cannot be purchased from the open market; *b)* it may be too late to start reverse engineering after a chip is on the market, as the market opportunity may have already expired; and *c)* a corresponding working chip may not be available if the adversary targets IP blocks or sub-modules.

Consequently, researchers have devised various *oracle-less* attacks on logic locking, including synthesis-based attacks [7], [8], testing-based attacks [8], [9], machine-learning-based attacks [10], [11] and structural attacks [12], [13]. These attacks do not require to access a working chip. However, these methods are either limited to specific use cases and locking schemes, or cannot guarantee termination and correctness. *Is there a logic decryption method as powerful as the SAT attack when a working chip is unavailable?*

TABLE I: Comparing the adversarial models of different logic decryption methods. A filled (*resp.* unfilled) circle indicates a resource required (*resp.* not required) by the adversary.

Method	Logic-locked Netlist	Working Chip	Functional Description	Formal Guarantee
Oracle-guided Attacks [2], [17], [18], [20], [21]	●	●	○	✓
Oracle-less Attacks [7]–[13]	●	○	○	✗
The DE2 Attack (Proposed Method)	●	○	●	✓

With the rapid growth of system-on-chip, heterogeneous architecture, and AI accelerator, *system-level modeling* is increasingly popular in the IC design flow. The system-level model constitutes *i)* an executable specification for the design team to drive the entire RTL development, *ii)* a golden reference model for the verification team to ensure that the RTL design conforms to the specification, and *iii)* a functional representation of the hardware that allows the software development team to reduce time to market. Unlike the locking key of the chip, the system-level model is distributed among multiple parties inside and outside the design house. Meanwhile, EDA vendors have released virtual prototyping tools such as Synopsys Virtualizer and Cadence Helium to support high-level functional specifications. Leading IP vendors, including Synopsys, ARM, and the RISC-V community, are offering compatible *transaction-level models* (TLMs) for their products [14]. Many of these models are made publicly accessible to attract customers. Furthermore, the *instruction-level abstraction* (ILA) modeling platform [15] has recently been proposed to facilitate the verification of processors and accelerators. An open-source database, ILDB [16], contains a wide variety of ILA models.

The main obstacle to adopt a high-level specification as the oracle is the timing problem. Existing sequential SAT attacks [17], [18] require the oracle to be cycle-accurate to the logic-locked netlist. However, the high-level models are typically loosely timed, approximately timed, or even untimed, versus their RTL implementations [19]. Additionally, the efficiency and scalability of sequential SAT attacks are questionable under these complex timing situations.

This paper presents DE2, a novel and rigorous sequential logic decryption method based on formal methods. Table I compares DE2 with existing logic decryption methods. DE2 takes a functional description instead of an oracle circuit to launch a sequential logic decryption. The functional description may contain limited or no timing information and may specify only the external I/O behaviors rather than the internal implementation details. DE2 first converts the functional specification to a netlist utilizing off-the-shelf high-level and logic synthesis tools. A clock cycle alignment between the logic-locked and the synthesized netlists is then automatically generated on the fly according to the I/O behaviors. Finally, it applies SAT-based

*Equal contribution.

† This work is partially supported by the National Science Foundation under grants 2113704 and 2148177.

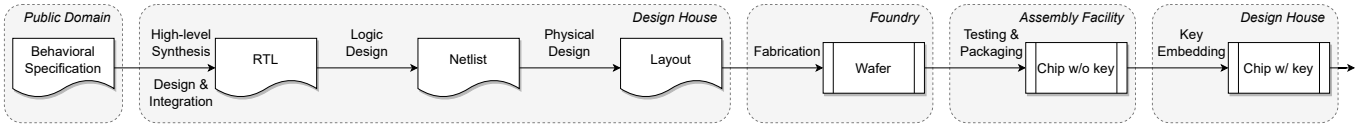


Fig. 1: IC design flow with logic locking.

constraint solving to extract a correct key from the model.

A crucial enabler of DE2 is a simple and flexible parameter synthesis attack against logic locking. Instead of duplicating the whole unrolled netlist, the parameter synthesis attack distills a *cube* (§II-A) of wrong keys from each I/O constraint. This attack can scale to large and hard benchmarks that demand deep unrolling, as it significantly reduces memory consumption and improves computational efficiency.

Our main contributions are:

- We discover that logic locking remains vulnerable to I/O attacks even when the adversary cannot obtain a working chip, challenging the common assumption in existing logic locking schemes;
- We propose LIM, a flexible and efficient sequential logic decryption algorithm outperforming state-of-the-art techniques;
- We devise DE2, an innovative SAT-based attack that can automatically align two non-cycle-accurate designs on the fly, enabling effective logic decryption without the need for a cycle-accurate oracle;
- We conduct extensive experiments to demonstrate the effectiveness and scalability of LIM and DE2 on a wide range of benchmark circuits.

II. Background

A. Preliminaries

A *combinational* logic-locked netlist C_e is defined as a tuple $\langle X, K, Y\rangle$, where X represents the primary inputs, K the key inputs, and Y the primary outputs. We use C_o to denote the oracle circuit corresponding to C_e. If a correct key K_c is inserted, $C_e(K_c)$ and C_o should exhibit the same behavior given any input sequence.

A variable or its negation is called a *literal*. A conjunction of literals is called a *cube*, and a disjunction of literals is called a *clause*. A clause is the negation of a cube and vise versa. A *pattern* is an assignment to all corresponding variables and it can be described as a cube. A *sequence* is a series of patterns. A Boolean formula is in conjunctive normal form (CNF) if it is a conjunction of clauses. A formula F *implies* another formula G, written as $F \Rightarrow G$, if all assignments satisfying F also satisfy G. I is an *implicant* of a formula F if I is a cube and $I \Rightarrow F$. I_p is a *prime implicant* if it is *minimal*, *i.e.,* dropping any literals from I_p will result in a non-implicant.

A sequence σ *stutters* at step k if it keeps the same value for indices k and $k+1$. For instance, the sequence $\langle a, b, c, c, \cdots\rangle$ stutters at step 3. Two sequences are *equivalent modulo stuttering* if they become identical after all stuttering steps are eliminated.

B. Problem Definition

Our attack requires that the functional specification and the original design of the circuit have the same externally visible behaviors modulo timing differences [22]:

Definition 1 (Observational equivalence)**.** *Two circuits are observationally equivalent if and only if their output sequences are equivalent modulo stuttering for every input sequence.*

Figure 2 shows a functional specification of Euclid's algorithm in a high-level language and the corresponding RTL implementation. For conciseness, we omitted key-related components in the RTL implementation. Because only one subtractor is allocated in the

```
while (x-y != 0)
{
 x = (x-y)>0 ? (x-y) : x;
 y = (y-x)>0 ? (y-x) : y;
}
out = x;
return;
```

(a) gcd_{spec}

```
always @(posedge clk) begin
 if (x-y != 0)
  x <= (x-y)>0 ? (x-y) : y
  y <= (x-y)>0 ? y : x
 else
  out <= x
end
```

(b) gcd_{impl}

Fig. 2: (a) The high-level *specification* and (b) the hardware *implementation* of Euclid's algorithm.

implementation, the two models are different in timing and have variable latencies with respect to each other. The adversary's goal is to infer a correct key for the logic-locked netlist:

Problem 1 (Non-cycle-accurate sequential logic decryption)**.** *Given a logic-locked netlist C_e and a corresponding reference model C_f, find a correct key K_c such that $C_e(K_c)$ and C_f are observationally equivalent.*

C. SAT Interface

We use $SAT[\psi]$ to denote a SAT query to formula ψ. It returns *satisfiable* if there exists an assignment to all variables in ψ such that ψ evaluates to true, and *unsatisfiable* otherwise. Furthermore, we use $SAT[\psi].model(X)$ to denote the value of variable X within a satisfiable assignment.

Many modern SAT solvers support unsat core extraction. We write an unsat core query as $SAT[\psi, \gamma]$, where ψ is a CNF formula and γ is a set of *assumption clauses*. If $\psi \wedge \gamma$ is satisfiable, both $SAT[\psi, \gamma]$ and the standard SAT query $SAT[\psi \wedge \gamma]$ will return a satisfiable assignment. However, if $\psi \wedge \gamma$ is unsatisfiable, $SAT[\psi, \gamma]$ will return an unsat core β in addition to the unsatisfiable result. β is a subset of γ such that $\psi \wedge \beta$ is still unsatisfiable. Some SAT solvers can ensure that β is minimal in most cases [23].

D. SAT Attack Algorithms

The combinational SAT attack algorithm [2] takes an *logic-locked netlist* C_e and a corresponding *oracle* circuit C_o as inputs. In each iteration, it queries an SAT solver for a differentiating input pattern (DIP) X_d: with such an input pattern, there exist two different key patterns K_1 and K_2 within the current model that produce two different output patterns Y_1 and Y_2. The algorithm then queries the oracle for the corresponding correct output Y_d. X_d and Y_d, together with a fresh copy of the logic-locked netlist, form a new *I/O constraint* on incorrect key patterns. The algorithm terminates when no more DIPs can be found. A correct key K_c can then be extracted from the model through another SAT query.

The combinational SAT attack can be thwarted if the scan-chain access to the flip-flops is disabled [24]. Besides, researchers have developed sequential logic locking schemes [25], [26]. They can be unlocked with an unlocking input sequence that guides the circuit from the *reset state* to the real *protected initial state*. Sequential logic decryption algorithms [18], [20] apply the combinational SAT attack on the unrolled netlist to alleviate these challenges. It treats the protected initial state as the key when attacking sequential logic locking schemes. The *unrolling diameter* k is increased in each iteration until one of the following termination conditions is

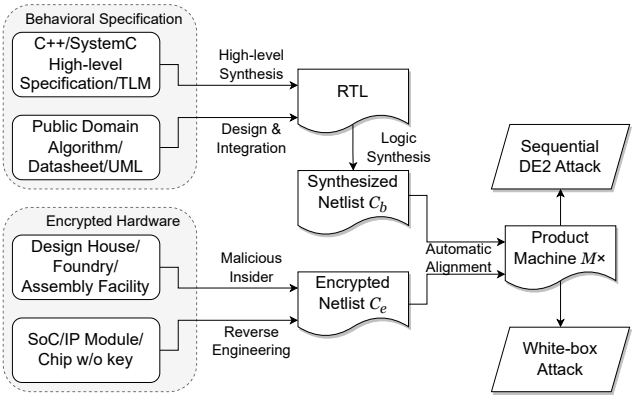


Fig. 3: The general workflow of DE2.

satisfied. *i) Unique Key:* only one key is remaining; *ii) Combinational Equivalence:* all the remaining keys are within the same equivalent class when the flip-flops are treated as primary inputs and outputs; *iii) Unbounded Model Checking:* an unbounded model checker cannot find a new differentiating input sequence.

III. The DE2 Attacking Algorithm

Figure 3 shows the general workflow of DE2. As mentioned in §I, DE2 assumes that the adversary cannot acquire an oracle circuit. Instead, the adversary is able to obtain a minimum functional specification of the victim circuit. This specification is automatically synthesized to an RTL design with a high-level synthesis tool (*e.g.,* Vivado HLS and Cadence Stratus). Afterwards, the RTL design is mapped to logic gates with a logic synthesis tool. In the remainder of this paper, we refer to the synthesized netlist as C_f and the expected behavior of the victim circuit as $C_e(K_c)$.

The functional specification may be untimed or have only limited timing information because it lacks internal implementation details. Additionally, the synthesis process can introduce uncertainties in timing. As a result, C_f and $C_e(K_c)$ are likely to be non-cycle-accurate. Furthermore, C_f could have a varying period with respect to $C_e(K_c)$, since a high-level synthesis tool can create a control logic that is fundamentally different from the one within C_e.

The DE2 algorithm (§III-C) brings up two main components to tackle the timing challenge. One component is an automatic alignment mechanism (§III-B). It attempts to insert stuttering steps on the fly to maintain the observational equivalence between C_f and $C_e(K_c)$. We formalize the observational equivalence constraints as CNF formulas so that they are compatible with SAT solvers. The other component is the LIM key condition synthesis algorithm (§III-A). With LIM, the size of the key condition is agnostic to the unrolling diameter, while the format of the key condition is irrelevant to the observational equivalence constraints. These benefits ensure feasibility and scalability for the DE2 algorithm. We also point out that DE2 enables new opportunities in traversing the error matrix of logic locking (§III-D).

A. LIM: A Parameter Synthesis Attack against Logic Locking

Each iteration in the SAT attack duplicates the logic-locked netlist to create a new I/O constraint. As a result, clauses pile up quickly within the SAT solver. It is observed that the execution time of each SAT query increases super-linearly with the number of DIPs [17]. KC2 [17] compresses the I/O constraints to speed up computation, but it still suffers from the drawbacks caused by netlist duplication.

We resort to formal verification to address this issue. We observed that almost every symbolic formal verification method [28]–[31] induces a corresponding logic decryption method. Table II shows

TABLE II: Symbolic verification vs. logic decryption.

	SAT	SMT	BMC	UMC	ParamSynth
Symbolic Verification	DPLL (1960)	Nelson (1979)	Unroll (2001)	ITP (2003)	IC3-based (2015)
Logic Decryption	[2] (2015)	[27] (2019)	[20] (2017)	[17], [18] (2019)	**LIM**

representative methods from both categories and the earliest year they were proposed. Recently, a new formal verification method called *parameter synthesis* has emerged [32], [33]. It aims to find the set of all correct patterns, given which a parameterized state transition system satisfies a safety property.

The SAT attack is essentially a covering process [34]: it terminates when I/O constraints eliminate exactly all wrong keys. This process is like parameter synthesis when the key variables are considered as the parameters of the circuit and functional equivalence to the oracle as the safety property. However, the adversary in the SAT attack has only *query access* to the oracle. Therefore, the current parameter synthesis methods cannot be used directly for logic decryption.

Algorithm 1 LIM: The Parameter Synthesis Attack Algorithm

1: **Input:** logic-locked netlist C_e, oracle circuit C_o
2: **Output:** a correct key pattern K_c
3: $W \leftarrow$ `True`
4: $M \leftarrow C_e(X, K_1, Y_1) \wedge C_e(X, K_2, Y_2)$
5: **while** $SAT[W \wedge M \wedge (Y_1 \neq Y_2)]$ **do**
6: $\quad X_d \leftarrow SAT[W \wedge M \wedge (Y_1 \neq Y_2)].model(X)$
7: $\quad Y_d \leftarrow C_o(X_d)$
8: $\quad K_w \leftarrow SAT[W \wedge C_e(X_d, K, Y) \wedge (Y \neq Y_d)].model(K)$
9: $\quad K_w^\circ \leftarrow SAT[W \wedge C_e(X_d, K, Y_d), \{K_w\}]$ ▷ *refer to §II-C*
10: $\quad W \leftarrow W \wedge \neg K_w^\circ$ ▷ *eliminate a generalized cube of wrong keys*
11: $K_c \leftarrow SAT[W].model(K)$

With all the observations above, we propose LIM (less-is-more), an algorithm that incrementally extracts information from the I/O constraints using parameter synthesis. LIM (Algorithm 1) maintains a CNF formula W to block all incorrect keys discovered so far (line 3). In each iteration, the algorithm finds a new incorrect key K_w (line 8), and generalize it to a cube to cover multiple wrong keys (line 9). Specifically, $\{K_w\}$ is a set of unit clauses: each unit clause corresponds to a literal in K_w. With $\{K_w\}$ enforced to K, the SAT query in line 9 must be unsatisfiable. Moreover, an SAT solver as described in §II-C should return a minimal unsat core K_w°, which is a prime implicant of the formula $\neg(W \wedge C_e(X_d, K, Y_d))$. In other words, K_w° is an irreducible cube distilled from both the current I/O constraint and all the previously generalized cubes. At the end of an iteration, the key condition W is updated by the negation of K_w°. Finally, when no more DIPs can be found, a correct key is extracted from the remaining keys within W (line 11).

Theorem 1. *The parameter synthesis attack algorithm will eventually terminate. Upon termination, formula W excludes exactly all incorrect keys.*

Proof. Termination part: In every iteration, at least one incorrect key will be discovered and conjoined to W. Meanwhile, there is only a finite number of incorrect keys.
Soundness part: A K_w produced in line 8 must be an incorrect key, because the output pattern Y generated by X_d and K is inconsistent with the reference output Y_d. By the same reasoning, all K_w° cubes contain only incorrect keys.
Completeness part: If an incorrect key is not yet excluded by W, the SAT query on line 5 will be satisfiable. In this case, the algorithm is not terminated, which conflicts with the condition. □

B. Automatic Alignment of Clock Cycles

Definition 1 states a criterion to determine whether two circuits are observationally equivalent: for any particular input sequence, the output sequences of the two circuits are identical after removing all stuttering steps. We observe that removing a stuttering step from one sequence is the *dual* of inserting a stuttering step to the other at the corresponding location. Inspired by studies on sequential formal equivalence checking [35], [36], we propose the following strategy to automatically enforce a sound equivalence relation to C_f and C_e:

Automatic Alignment Mechanism. *Whenever C_f's output pattern changes, stall C_f until C_e's output pattern also changes. Likewise, let C_e wait for C_f whenever it is faster.*

We use the example in Figure 2 to illustrate this mechanism. Suppose the initial values of x and y are 6 and 10, and only the final states of both designs are observable. This mechanism will insert stuttering steps before the final state of gcd_{spec}, because only then the output pattern is changed. The execution traces of gcd_{spec} and gcd_{impl} are $\langle(6,10),(6,4),(2,4),(2,4),(2,4),(2,4),(2,2)\rangle$ and $\langle(6,10),(10,6),(4,6),(6,4),(2,4),(4,2),(2,2)\rangle$, respectively. For another example, suppose gcd_{spec} and gcd_{impl} are modified so that the smaller value between x and y is always observable in every clock cycle. In that case, stuttering steps are periodically inserted to gcd_{spec} to ensure equivalence. The execution traces of gcd_{spec} and gcd_{impl} become $\langle(6,10),(6,10),(6,4),(6,4),(2,4),(2,4),(2,2)\rangle$ and $\langle(6,10),(10,6),(4,6),(6,4),(2,4),(4,2),(2,2)\rangle$, respectively. It can be seen that alignment is deterministic given the data and the observability condition, thus significantly reducing the search space of DE2. The following lemma proves the correctness of this mechanism.

Lemma 2. *C_f and C_e are observationally equivalent if and only if the automatic alignment mechanism produces identical output sequences for any input sequences.*

Proof. Only If part: To show that the output sequences of the two circuits are identical, we split the output sequence of the slower circuit into segments: the first step in a segment is a *critical step* (whose output pattern is different from the previous step), and the rest of the steps in that segment are *stuttering steps* (whose output pattern is the same as the previous step). The above mechanism is to process the segments of the faster circuit's output sequence successively. For every segment, it attaches dummy stuttering steps until it matches the corresponding segment of the slower circuit.
If part: After removing all stuttering steps, two initially identical sequences are still identical. Hence, by Definition 1, two circuits are observationally equivalent if their output sequences are always identical given any input sequences. □

We implement this mechanism by constructing a *product machine* consisting of C_f and C_e. This machine detects whether exactly one circuit will change its output value in the next clock cycle. If so, that circuit will be stalled in the next clock cycle. If not, both circuits can continue moving forward. In our implementation, stalling is realized by adding a multiplexer and a feedback path for every flip-flop. Furthermore, we add a *miter* to the product machine to determine whether the output values of the two circuits are always identical. The miter outputs `True` if they output *distinct* patterns at any clock cycle. Finally, we add a bounded fairness constraint [37], which enforces both circuits to move forward at least once in every n consecutive steps. To check whether the two circuits are observationally equivalent, one can call a model checker on the product machine, and use the miter output as the safety property.

Our approach is more suitable for logic decryption than sequential equivalence checking techniques such as Kairos [35] for the following reasons: *a)* it does not require *valid* or *ready* signals for synchronization; *b)* it will not produce a false negative when either of the circuits is trapped in a deadlock state; *c)* it does not rely on special techniques like clock gating, so the generated product machine can be easily encoded as CNF formulas.

C. The DE2 Algorithm

Algorithm 2 The DE2 Algorithm

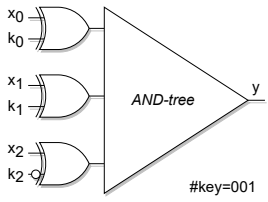
1: **Input:** logic-locked netlist C_e, synthesized netlist C_d
2: **Output:** a correct key pattern K_c
3: $k \leftarrow 1$
4: $W \leftarrow$ `True`
5: $M_s \leftarrow C_e(X,K,Y_1) \times C_d(X,Y_2)$
▷ *construct the product machine with automatic alignment*
6: **while** `True` **do**
7: $M_u \leftarrow unroll(M_s,k)$
8: **while** $SAT[W \wedge M_u \wedge (Y_1 \neq Y_2)]$ **do**
9: $X_d \leftarrow SAT[W \wedge M_u \wedge (Y_1 \neq Y_2)].model(X)$
10: $K_w \leftarrow SAT[W \wedge M_u \wedge (X = X_d) \wedge (Y_1 \neq Y_2)].model(K)$
11: $K_w^\circ \leftarrow SAT[W \wedge M_u \wedge (X = X_d) \wedge (Y_1 = Y_2), \{K_w\}]$
12: $W \leftarrow W \wedge \neg K_w^\circ$
13: $K_p \leftarrow SAT[W].model(K)$
14: **if** $UMC[M_s \wedge (K = K_p) \wedge (Y_1 \neq Y_2)]$ = `False` **then**
15: $K_c \leftarrow K_p$
16: **break**
17: $k \leftarrow k+1$ ▷ *increase the unrolling diameter*
18: **return** K_c

With a correct key K_c, the miter should never output a `True` for any input sequences. In other words, the correct key can be extracted from the model if a `False` is asserted for all input sequences ($\exists K_c$ *s.t.* $\forall$ input sequences: miter's output = `False`). Unfortunately, due to the quantifier alternation [36] in the expression, immediately extracting K_c is still infeasible.

Hence, the DE2 algorithm (Algorithm 2) iteratively blocks incorrect keys until it finds a correct one. It builds a product machine M_s with the automatic alignment mechanism (line 5). M_u denotes the unrolled product machine with an initial unrolling diameter of 1. The inner loop of the algorithm (line 8-12) is similar to LIM. The key condition W is strengthened until it blocks exactly all incorrect keys activating the miter ($Y_1 \neq Y_2$) up to the current unrolling diameter. However, checking for termination is time-consuming [21] because the required unrolling diameter is unknown. In this regard, we develop a new condition, trial-and-error (*TE*), for improved efficiency. This condition first extracts a *potentially* correct key K_p from W (line 13). It then queries an unbounded model checker (*UMC*) for the correctness of K_p (line 14). Since the search space is now limited to just a single key, the execution time of model checking is greatly reduced.

Theorem 3. *The DE2 algorithm will eventually terminate and return a correct key.*

Proof. Termination part: In every iteration, at least one incorrect key will be discovered and conjoined to W. Hence, the algorithm will eventually terminate.
Correctness part: From Lemma 2 and Theorem 1, the algorithm must return a correct key if a K_c exists such that C_f and $C_e(K_c)$ are observationally equivalent. □



(a)

x \ k	000	001	010	011	100	101	110	111
000	0	0	0	0	0	0	1	0
001	0	0	0	0	0	0	0	1
010	0	0	0	0	1	0	0	0
011	0	0	0	0	0	1	0	0
100	0	0	1	0	0	0	0	0
101	0	0	0	1	0	0	0	0
110	1	0	0	0	0	0	0	0
111	0	1	0	0	0	0	0	0

(b)

Fig. 4: (a) A 3-input AND-tree protected by input-flip locking [5]. (b) The corresponding error matrix. Shadowed entries with underscores have incorrect outputs.

D. White-box Attack Against Logic Locking

The error matrix is a powerful tool for analyzing the complexity of an attacking algorithm against a logic locking scheme. Each row in the error matrix represents an input pattern, each column represents a key pattern, and each entry represents whether the output pattern of the logic-locked netlist deviates from the correct output pattern. In the original SAT attack, every query to the oracle circuit can reveal only one row of the error matrix. Existing logic locking schemes [3], [4] exploit this limitation to thwart the SAT attack. They ensure that the error rate is exponentially small in the number of input patterns. Formally speaking, for every column representing an incorrect key, there is only a limited number of incorrect entries.

DE2 gives the adversary another degree of freedom when traversing the error matrix. We use a case study to demonstrate the capability of the white-box attack. Figure 4(b) plots the error matrix of an AND-tree protected by input-flipping locking. Once the attacker discovers the *dominant row* ($X = 111$), it will immediately exclude all incorrect keys. Assuming an SAT solver that selects the next row uniformly at random [2], the attacking algorithm must search through half of the rows on average until it finds the dominant one. Facilitated by the automatic alignment mechanism, DE2 has almost white-box access to C_e, which allows it to quickly locate the dominant row using SAT-based *model enumeration*. Specifically, it randomly selects a row, K^*, and queries $SAT[M_s \wedge (K = K^*) \wedge (Y_1 \neq Y_2)].model(X)$ for a new assignment X^*. If such an assignment does not exist, K^* must be correct. Otherwise, it conjoins $\neg X^*$ to the formula and repeats the above query. Because the number of satisfiable rows for a given K^* is *exponentially* smaller than the number of feasible DIPs in this example, the white-box attack can be significantly faster than the SAT attack. The white-box attack can be adapted to defeat other logic locking schemes.

IV. Evaluations

A. Experimental Setup

We built a prototype for LIM and DE2 in Python. We chose Boolector [38] and Z3 [39] as the backend SMT solvers, because we observed that they are more efficient on hardware benchmarks in terms of SAT solving and unsat core extraction, respectively. We employed the property-directed reachability command *pdr* [40] from Berkeley ABC for unbounded model checking. All experiments are conducted on a Linux machine with a 3.2GHz CPU, and every instance was executed on a single thread. We set a memory limit of 4GiB and a timeout limit of 3,600 seconds for all experiments. We assess the effectiveness and efficiency of LIM and DE2 by answering the following research questions:

RQ1. Does LIM, the core module of DE2, have a better performance over existing sequential logic decryption algorithms?

RQ2. Is DE2 realistically applicable to the situation where only a functional description is available?

TABLE III: A comparison of execution time (seconds) for KC2, RANE, and LIM on *XOR*-locked sequential benchmark circuits.

Overhead	5%			10%			15%		
Method	KC2	RANE	LIM	KC2	RANE	LIM	KC2	RANE	LIM
s208	–	–	–	–	0.6	1.8	–	–	–
s298	0.1	0.7	1.6	–	–	–	1.2	2.4	4.7
s344	0.4	0.6	1.6	0.3	0.9	2.7	0.5	1.1	3.6
s349	0.1	0.4	1.1	0.1	0.5	1.5	1.0	1.1	4.1
s382	106	120	47.1	104	63.1	34.2	–	–	–
s386	0.1	0.8	1.0	0.2	0.9	2.3	0.3	0.7	4.1
s400	88.6	62.6	27.0	–	–	278	–	–	2772
s444	–	3507	–	–	–	–	–	–	–
s510	1.8	3.6	12.1	0.7	1.4	13.4	39.1	34.7	861
s526	–	–	–	–	1451	2839	–	–	–
s526n	80.8	–	–	–	92.9	2014	–	–	–
s641	0.5	1.2	3.1	0.4	–	4.9	–	–	–
s713	0.4	2.6	3.2	1.5	3.3	7.1	–	–	–
s820	3.1	4.7	8.7	6.8	6.9	12.4	16.6	10.8	22.0
s832	1.8	2.9	7.2	7.3	6.6	15.3	12.5	9.7	27.3
s838	–	–	–	–	–	–	–	–	–
s953	–	–	12.8	–	–	18.4	–	–	57.6
s1196	0.5	3.7	3.7	1.1	–	6.7	6.0	5.2	15.8
s1238	0.5	1.8	3.5	1.3	3.8	11.8	2.0	4.1	13.3
s1423	–	1912	575	–	–	1884	–	–	–
s1488	8.1	10.1	36.0	22.2	14.1	99.2	187	44.9	229
s1494	4.0	5.5	21.8	94.4	28.2	125	91.8	19.9	187
s5378	–	–	–	–	–	–	–	–	–
s9234	–	–	–	–	–	–	–	–	–

B. LIM for Standard Sequential Logic Decryption

To answer **RQ1**, we compared LIM against the state-of-the-art sequential logic decryption algorithms, including the KC2 command [17] in NEOS and the RANE decryption suite [18]. We used the default settings of the oracle-guided sequential SAT attack for both algorithms. Since neither of them selects *UMC* (§II-D) as a termination condition, we also disabled *UMC* for LIM to foster fair comparisons. We incremented the rolling diameter by 1 in each step and set an upper limit of 100 steps for all algorithms.

We used the ISCAS'89 [41] sequential benchmark circuits for our evaluations. We locked the circuits with two prevalent logic locking schemes: *i)* randomly inserting key-controlled *XOR* gates to the combinational part of a circuit [2], and *ii)* the HARPOON sequential logic locking scheme [25], which requires an unlocking input sequence to steer a circuit from its reset state to its actual initial state. We constructed three logic-locked instances with different locking overheads for each benchmark circuit. We followed the same methodology as KC2 [17], which represents key sizes as the percentage increases in gate counts (overheads). In particular, we used the default CMOS cells library in Yosys [42] to measure the gate counts after technology mapping. For instance, a 15% overhead corresponds to 82 bits for *s1196* and 443 bits for *s5378*, respectively.

Table III compares the execution time of the three algorithms on *XOR*-locked instances. It appears that these algorithms have similar overall performances. Due to the unrolling, sequential logic

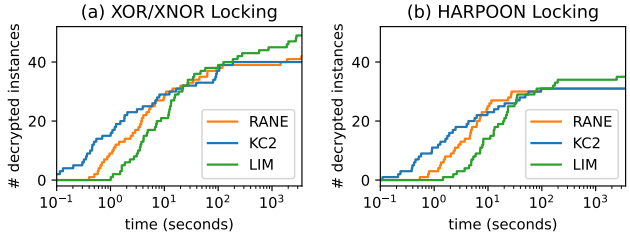


Fig. 5: Comparisons of the number of decrypted instances over time by different methods.

TABLE IV: Execution time (seconds) of DE2 on *XOR*-locked sequential benchmark circuits.

Overhead	5%	10%	15%	Overhead	5%	10%	15%
s208	2.4	31.9	605	s713	62.0	998	820
s298	0.9	18.3	901	s820	13.8	77.2	706
s344	1.0	6.3	683	s832	17.8	43.0	409
s349	0.9	10.8	1566	s838	–	–	–
s382	365	–	1079	s953	38.8	1350	–
s386	0.6	114	131	s1196	619	16.4	601
s400	296	1170	–	s1238	5.0	11.4	706
s444	155	–	–	s1423	–	–	–
s510	207	174	–	s1488	46.0	146	–
s526	–	–	–	s1494	339	804	–
s526n	–	–	–	s5378	–	–	–
s641	3.9	634	669	s9234	–	–	–

TABLE V: Statistics of high-level synthesized benchmark circuits.

Circuit	#PI	#PO	#FF	#Gate
euclid	8	5	11 - 22	231 - 315
gcd	8	5	11 - 37	255 - 766
barcode	9	10	17 - 35	279 - 591
counter	6	5	17 - 34	357 - 469
numeric	12	13	15 - 30	506 - 623
fuzzy	13	5	33 - 51	669 - 963
diffeq	20	13	27 - 42	864 - 1073
ellipf	32	33	33 - 63	1339 - 1433
kalman	22	9	14 - 69	1507 - 1909
wavef	32	33	72 - 108	1788 - 2183

decryption algorithms generally cannot scale to large sequential circuits like *s5378* and *s9234*. Additionally, Figure 5 compares the three algorithms by the number of decrypted instances over time. It displays how many instances out of 72 can be decrypted (the vertical axis) by each algorithm if all instances are assigned the same timing budget (the horizontal axis). LIM is initially slower than the other two, but it eventually stands out. After 3,600 seconds, it decrypts 23% and 17% more instances than KC2 and RANE, respectively. Due to the page limit, we omit the table of the HARPOON experiments. LIM can decrypt 13% more instances than either of the other algorithms within 3,600 seconds on HARPOON-locked instances. Hence, we conclude that LIM has a better performance than the state-of-the-art logic decryption algorithms for higher timing budgets.

C. DE2 for Description-guided Logic Decryption

To answer **RQ2**, we evaluate the capability of DE2 on a variety of benchmarks. We used the same settings as §IV-B except that we restored *UMC* as a termination condition.

For every instance in Table III, we inserted a stalling logic to extend its period by a constant. Existing I/O attacks cannot be applied to this scenario without exactly knowing this constant, because the oracle circuit C_o and the logic-locked netlist C_e are different in timing. Table IV summarizes the results of DE2 on these instances. DE2 can decrypt 64% of the instances within 3,600 seconds, and the average execution time of decrypted instances is 694.5 seconds. Meanwhile, as shown in Table III, LIM can decrypt 68% of the instances within 3,600 seconds, and the average execution time of decrypted instances is 41.6 seconds. Although the results in Table IV are not directly comparable with those in Table III, we emphasize that most instances solvable by LIM are also solvable by DE2. We also observed that the *TE* strategy significantly reduces the execution time of *UMC*. This improvement is vital because *UMC* can dominate the total execution time on those instances which require a large unrolling diameter.

We designed another experiment to assess DE2 in a more realistic setting. We used a high-level synthesis tool, Vivado HLS, to synthesize a set of behavioral-level designs from the HLSynth benchmark suite [43] to RTL designs. We utilized HLS pragmas, including pipeline, initiation interval, resource allocation, latency, unroll, flatten, partition, balance, *etc.*, to control an RTL design's timing. For each behavioral design, we generated 4 RTL designs with distinct latencies. Afterwards, we applied Yosys [42] to map them to gate-level netlists. We treated the netlist with the smallest latency as the reference netlist C_d. Each of the remaining netlists of the same behavioral design is logic-locked by the *XOR* locking scheme with different locking overheads. As such, we constructed 9 instances of the logic-locked netlist C_e for every behavioral design. Table VI shows the execution time of DE2 on realistic test cases. Given a timing budget of 3,600 seconds, DE2 successfully decrypts 73%, 67%, and 40% of the instances that are logic-locked with 3%, 5%, and 10% of overheads, respectively. All these results demonstrate the effectiveness and usefulness of DE2 under the new adversarial model.

TABLE VI: Execution time (seconds) of DE2 on high-level synthesized sequential benchmark circuits.

Latency	Small			Medium			Large		
Overhead	3%	5%	10%	3%	5%	10%	3%	5%	10%
euclid	0.7	97.8	6.3	1.6	8.9	–	19.9	1547	–
gcd	23.0	–	–	504	–	1.2	11.7	3569	–
barcode	2.0	907	–	7.0	42.4	181	–	5.1	–
counter	191	13.1	7.8	3.4	14.2	6.0	2.0	43.7	3.2
numeric	2.9	1437	48.7	2.0	–	203	3.6	1.3	–
fuzzy	34.7	–	–	1.8	–	–	–	45.8	–
diffeq	2.3	61.4	140	2.6	3.1	–	–	–	–
ellipf	12.9	13.3	257	640	–	–	–	5.5	5.7
kalman	130	8.6	–	–	9.1	–	–	–	–
wavef	–	–	–	239	219	–	–	–	142

Finally, we compared the white-box DE2 (§III-D) against the standard DE2 (§III-C) to showcase the advantage of traversing the error matrix from both directions. The benchmark circuits are logic-locked with the double-flip locking scheme [44] which guarantees exponentially low error rates. As depicted in Figure 6, the highest speedup achieved by the white-box DE2 is 144×. We also noticed that the improvements are more prominent with larger input sizes. This is because there is only a limited number of incorrect entries in each column of the error matrices. Notice that many other recent defense schemes with exponentially low error rates [1] are also vulnerable to this white-box attack.

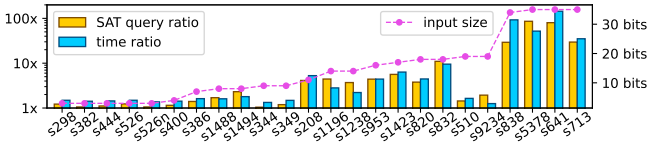


Fig. 6: White-box DE2 vs. standard DE2 on logic-locked benchmark circuits with exponentially low error rates. The left coordinate measures the acceleration ratio, and the right coordinate measures the number of input bits.

V. Conclusion

This paper investigates whether logic locking is susceptible to I/O attacks without a cycle-accurate oracle circuit. Our study reveals that an adversary can still launch a sequential SAT attack using a functional specification of the victim circuit. This result calls for new efforts to reassess existing logic locking techniques and to enhance their security. The proposed DE2 algorithm provides a method to evaluate the security of logic locking under the new adversarial model.

REFERENCES

[1] H. M. Kamali, K. Z. Azar, F. Farahmandi, and M. Tehranipoor, "Advances in logic locking: Past, present, and prospects," *Cryptology ePrint Archive*, 2022.

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

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

[4] Y. Xie and A. Srivastava, "Anti-sat: Mitigating sat attack on logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 2, pp. 199–207, 2018.

[5] H. Zhou, A. Rezaei, and Y. Shen, "Resolving the trilemma in logic encryption," in *ICCAD 2019*, pp. 1–8.

[6] K. Zamiri Azar, H. Mardani Kamali, H. Homayoun, and A. Sasan, "Threats on logic locking: A decade later," in *GLSVLSI 2019*, pp. 471–476.

[7] A. Alaql, D. Forte, and S. Bhunia, "Sweep to the secret: A constant propagation attack on logic locking," in *AsianHOST 2019*, pp. 1–6.

[8] L. Li and A. Orailoglu, "Piercing logic locking keys through redundancy identification," in *DATE 2019*, pp. 540–545.

[9] D. Duvalsaint, X. Jin, B. Niewenhuis, and R. Blanton, "Characterization of locked combinational circuits via atpg," in *ITC 2019*, pp. 1–10.

[10] P. Chakraborty, J. Cruz, and S. Bhunia, "Sail: Machine learning guided structural analysis attack on hardware obfuscation," in *AsianHOST 2018*, pp. 56–61.

[11] L. Alrahis, S. Patnaik, M. Shafique, and O. Sinanoglu, "Omla: An oracle-less machine learning-based attack on logic locking," *TCAS II*, vol. 69, no. 3, pp. 1602–1606, 2021.

[12] Y. Zhang, P. Cui, Z. Zhou, and U. Guin, "Tga: An oracle-less and topology-guided attack on logic locking," in *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop*, 2019, pp. 75–83.

[13] A. Jain, Z. Zhou, and U. Guin, "Taal: tampering attack on any key-based logic locked circuits," *TODAES*, vol. 26, no. 4, pp. 1–22, 2021.

[14] Synopsys, "Virtualizer models," https://www.synopsys.com/verification/virtual-prototyping/virtualizer-models.html, 2024.

[15] B.-Y. Huang, H. Zhang, A. Gupta, and S. Malik, "Ilang: A modeling and verification platform for socs using instruction-level abstractions," in *TACAS 2019*, 2019, pp. 351–357.

[16] H. Lu *et al.*, "The ila model database," https://github.com/Princeton University/IMDb, 2021.

[17] K. Shamsi, M. Li, D. Z. Pan, and Y. Jin, "Kc2: Key-condition crunching for fast sequential circuit deobfuscation," in *DATE 2019*, pp. 534–539.

[18] S. Roshanisefat, H. Mardani Kamali, H. Homayoun, and A. Sasan, "Rane: An open-source formal de-obfuscation attack for reverse engineering of logic encrypted circuits," in *GLSVLSI 2021*.

[19] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2003, pp. 19–24.

[20] M. El Massad, S. Garg, and M. Tripunitara, "Reverse engineering camouflaged sequential circuits without scan access," in *ICCAD 2017*, pp. 33–40.

[21] Y. Hu, Y. Zhang, K. Yang, D. Chen, P. A. Beerel, and P. Nuzzo, "Fun-sat: Functional corruptibility-guided sat-based attack on sequential logic encryption," in *HOST 2021*, pp. 281–291.

[22] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, 1991.

[23] N. Dershowitz, Z. Hanna, and A. Nadel, "A scalable algorithm for minimal unsatisfiable core extraction," in *SAT 2006*, pp. 36–41.

[24] J. Da Rolt, G. Di Natale, M.-L. Flottes, and B. Rouzeyre, "New security threats against chips containing scan chain structures," in *HOST 2011*, pp. 110–110.

[25] R. S. Chakraborty and S. Bhunia, "Harpoon: An obfuscation-based soc design methodology for hardware protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1493–1502, 2009.

[26] A. R. Desai, M. S. Hsiao, C. Wang, L. Nazhandali, and S. Hall, "Interlocking obfuscation for anti-tamper hardware," in *Proceedings of the eighth annual cyber security and information intelligence research workshop*, 2013, pp. 1–4.

[27] 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.

[28] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM (JACM)*, vol. 7, no. 3, pp. 201–215, 1960.

[29] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 2, pp. 245–257, 1979.

[30] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal methods in system design*, vol. 19, no. 1, pp. 7–34, 2001.

[31] K. L. McMillan, "Interpolation and sat-based model checking," in *CAV 2013*, pp. 1–13.

[32] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Parameter synthesis with ic3," in *FMCAD 2013*, pp. 165–168.

[33] S. Ben-David, B. Sterin, J. M. Atlee, and S. Beidu, "Symbolic model checking of product-line requirements using sat-based methods," in *ICSE 2015*, vol. 1, pp. 189–199.

[34] A. Petkovska, A. Mishchenko, D. Novo, M. Owaida, and P. Ienne, "Progressive generation of canonical sum of products using a sat solver," in *IWLS 2016*.

[35] L. Piccolboni, G. Di Guglielmo, and L. P. Carloni, "Kairos: Incremental verification in high-level synthesis through latency-insensitive design," in *FMCAD 2019*, pp. 105–109.

[36] Y. Li, G. Zhao, Y. He, and H. Zhou, "Se3: Sequential equivalence checking for non-cycle-accurate design transformations," in *DAC 2023*.

[37] N. Dershowitz, D. Jayasimha, and S. Park, "Bounded fairness," *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, pp. 304–317, 2003.

[38] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2, btormc and boolector 3.0," in *CAV 2018*, pp. 587–595.

[39] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS 2008*, pp. 337–340.

[40] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *FMCAD 2011*, pp. 125–134.

[41] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *ISCAS 1989*, pp. 1929–1934.

[42] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Austrochip 2013*, p. 97.

[43] P. R. Panda and N. D. Dutt, "1995 high level synthesis design repository," in *ISSS'95*, pp. 170–174.

[44] M. Yasin, A. Sengupta, B. C. Schafer, Y. Makris, O. Sinanoglu, and J. Rajendran, "What to lock? functional and parametric locking," in *GLSVLSI*, 2017, pp. 351–356.