HOLL: Program Synthesis for Higher Order Logic Locking

Gourav Takhar[1]($\boxtimes$), Ramesh Karri[2], Christian Pilato[3], and Subhajit Roy[1]

[1] Indian Institute of Technology Kanpur, Kanpur, India.
{tgourav,subhajit}@cse.iitk.ac.in
[2] New York University, New York, NY, USA. rkarri@nyu.edu
[3] Politecnico di Milano, Milan, Italy. christian.pilato@polimi.it

Abstract. Logic locking "hides" the functionality of a digital circuit to protect it from counterfeiting, piracy, and malicious design modifications. The original design is transformed into a "locked" design such that the circuit reveals its correct functionality only when it is "unlocked" with a *secret* sequence of bits—the *key* bit-string. However, strong attacks, especially the *SAT attack* that uses a SAT solver to recover the key bit-string, have been profoundly effective at breaking the locked circuit and recovering the circuit functionality.

We lift logic locking to *Higher Order Logic Locking (HOLL)* by hiding a higher-order *relation*, instead of a key of independent values, challenging the attacker to discover this *key relation* to recreate the circuit functionality. Our technique uses *program synthesis* to construct the locked design and synthesize a corresponding *key relation*. HOLL has low overhead and existing attacks for logic locking do not apply as the entity to be recovered is no more a value. To evaluate our proposal, we propose a new attack (*SynthAttack*) that uses an inductive synthesis algorithm guided by an operational circuit as an input-output oracle to recover the hidden functionality. SynthAttack is inspired by the SAT attack, and similar to the SAT attack, it is *verifiably correct*, i.e., if the correct functionality is revealed, a verification check guarantees the same. Our empirical analysis shows that SynthAttack can break HOLL for small circuits and small key relations, but it is ineffective for real-life designs.

Keywords: Logic Locking · Program Synthesis · Hardware Security.

1 Introduction

High manufacturing costs in advanced technology nodes are pushing many semiconductor design houses to outsource the fabrication of integrated circuits (IC) to third-party foundries [26, 43]. A *fab-less* design house can increase the investments in the chip's intellectual property, while a single foundry can serve multiple companies. However, this globalization process introduces security threats in the supply chain [25]. A malicious employee of the foundry can access and reverse engineer the circuit design to make illegal copies. Logic locking [45] alters the

chip's functionality to make it unusable by the foundry. This alteration depends on a *locking key* that is re-inserted into the chip in a trusted facility, after fabrication. The locking key is, thus, the "secret", known only to the design house. Logic locking assumes that the attackers have no access to the key but they may have access to a functioning chip (obtained, for example, from the legal/illegal market). However, logic locking has witnessed several attacks that analyze the circuit and attempt key recovery [31, 44, 49, 60].

In this paper[4], we combine the intuitions from logic locking, program synthesis, and programmable devices to design a new locking mechanism. Our technique, called *higher order logic locking* (HOLL), locks a design with a *key relation* instead of a sequence of independent *key bits*. HOLL uses *program synthesis* [3, 51] to translate the original design into a locked one. Our experiments demonstrate that HOLL is fast, scalable, and robust against attacks. Prior attacks on logic locking, like the SAT attack [52], are not practical for HOLL. Since the functionality of the key-relation is completely missing, attackers cannot simply make propositional logic queries to recover the key (like [44, 52, 18]). There are variants of logic locking, like TTLock [62] and SFLL [61], that attempt to combat SAT attacks [52]. However, these techniques use additional logic blocks (comparison and restoration circuits) which makes them prone to attacks via structural and functional analysis on this additional circuitry [48]. HOLL is resilient to such techniques as it exposes only a programmable logic that does not leak any information related to the actual functionality to be implemented.

In contrast to logic locking, attacking HOLL requires solving a second-order problem (see §8 for a detailed discussion on this). To assess the security of our method, we design a new attack, *SynthAttack*, by combining ideas from SAT attack [52] and inductive program synthesis [51]. SynthAttack employs a counterexample guided inductive synthesis (CEGIS) procedure guided via a functioning instance of the circuit as an input-output oracle. This attack constructs a synthesis query to discover key relations that invoke *semantically distinct functional behaviors* of the locked design—witnesses to such relations, referred to as *distinguishing inputs*, act as counterexamples to drive inductive learning. When the locked design is verified to have unique functionality, the attack is declared successful, with the corresponding provably-correct key relation.

Our experimental results (§6) show that the time required by an attacker to recover the key relation for a given set of distinguishing inputs (*attack time*) increases exponentially with the size of key relation. While the attacker may be able to recover key relations for small HOLL-locked circuits with small key relations, larger circuits are robust to SynthAttack. For example, for the `des` benchmark, the asymmetry between HOLL defense and SynthAttack is large; while HOLL can lock this design in less than `100` seconds, the attack cannot recover the design even within four days for a key relation that increases the area overhead of the IC by only `1.2%`. Further, the attack time required to unlock the designs increase exponentially with the complexity of the key relation.

The key relation can be implemented with reconfigurable or programmable

---

[4] An extended version [54] of this paper is also available.

$t_0 = x_0 \wedge x_2;$
$t_1 = \boxed{x_3 \wedge t_0}$

$t_2 = \boxed{(x_1 \wedge t_0)}$
$y_0 = x_0 \oplus x_2$
$y_2 = (x_1 \wedge x_3) \vee t_2 \vee t_1$
$y_1 = t_0 \oplus x_1 \oplus x_3$

(a) Original circuit

$t_0 = x_0 \wedge x_2$
$t_1 = \boxed{(x_0 \wedge (r_4 \oplus r_2) \wedge x_3)}$

$t_2 = \boxed{(x_0 \wedge r_3)}$
$y_0 = x_0 \oplus x_2$
$y_2 = (x_1 \wedge x_3) \vee t_2 \vee t_1$
$y_1 = t_0 \oplus x_1 \oplus x_3$

(b) Locked circuit

$\{(r_0 \leftrightarrow x_1),$
$(r_1 \leftrightarrow x_2),$
$(r_2 \leftrightarrow rand),$
$(r_3 \leftrightarrow r_0 \wedge r_1),$
$(r_4 \leftrightarrow r_1 \oplus r_2)\}$

(c) Key relation

Fig. 1: HOLL on a 2-bit Adder.

devices, like programmable array logic (PAL) or embedded finite-programmable gate array (eFPGA). For example, eFPGA, essentially an IP core embedded into an ASIC or SoC, is becoming common in modern SoCs [2] and has been shown to have high resilience against bit-stream recovery [7].

Our contributions are:

- We propose a novel IP protection strategy, called *higher order logic locking* (HOLL), that uses program synthesis;
- To evaluate the security offered by HOLL, we propose a strong adversarial attack algorithm, SynthAttack;
- We build tools to apply HOLL and SynthAttack to combinational logic;
- We evaluate HOLL on cost, scalability, and robustness;

2 HOLL Overview

2.1 Threat Model: the Untrusted Foundry

We focus on the threat model where the attacker is in the foundry [45, 46] to which a fab-less design house has outsourced its IC fabrication. Such an attacker has access to the IC design and the (locked) GDSII netlist which can be reverse-engineered. Also, if the attacker can access a working IC (e.g., by procuring an IC from the open market or a discarded IC from the gray market), they can leverage the functional IC's I/O behavior as a black-box oracle. However, we assume the attacker *cannot* extract the *bitstream*, i.e. the correct sequence of configuration bits, from the device. This can be achieved with encryption techniques when the bitstream is not loaded into the device. Also, anti-readback solutions can prevent the attacker from reading the bitstream from the device. The parameters used to synthesize the *key relation* and the *locked circuit* (like the domain-specific language (DSL), budget etc.) are not known to the attacker (see §8).

2.2 Defending with *HOLL*

Consider a hardware circuit $Y \leftrightarrow \varphi(X)$, where X and Y are the inputs and outputs, respectively. HOLL uses a higher-order lock—a *secret* relation (ψ) among a certain number of additional *relation* bits R. We refer to ψ as the **key relation**.

Fig. 1a shows an example of a 2-bit adder with input X ($\{x_1x_0, x_3x_2\}$) and output Y ($y_2y_1y_0$). The circuit is locked by transforming the original expressions

(marked in blue) in Fig. 1a to the *locked* expressions (marked in red) in Fig. 1b. The locked expressions use the additional *relation* bits r_2, r_3, and r_4, enabling that this *locked design* $\hat{\varphi}(X, R)$ functions correctly when the secret relation ψ (Fig. 1c) is installed. The relation ψ establishes the correct relation between the **relation bits** R. The key relation can be excited by circuit inputs (like in r_0 and r_1,), constants, or random bits (e.g., from system noise, etc.); for example, the value *rand* in Fig. 1c represents the random generation of a bit (0 or 1) assigned to r_2. The "output" from the key relation are bits r_3 and r_4 that must satisfy the relational constraints enforced by the key relation.

For the sake of simplicity, in the rest of the paper, we assume the relation bits are drawn only from the inputs X of the design. We will attempt to infer key relations of the form $\psi(X, R)$. The reader may assume the value *rand* of in Fig. 1c to be a constant value (say 0) to ease the exposition.

As $\hat{\varphi}$ also consumes the *relation bits* R, HOLL transforms the original circuit $Y \leftrightarrow \varphi(X)$ into a *locked* circuit $Y \leftrightarrow \hat{\varphi}(X, R)$ such that the locked circuit functions correctly if the key relation $\psi(X, R)$ is satisfied. In other words, HOLL is required to preserve the semantic equivalence between the original and locked designs ($\varphi = \hat{\varphi} \wedge \psi$). Note that it only imposes constraints on the input-output functionality of the circuits, not on the generated values of internal gates. For example, in Fig. 1b, the value of t_1 may be different from the one in the original design (Fig. 1a), but the final output y_2 is equivalent to the original adder.

This approach has analogies with the well-known *logic locking* solution [10, 37, 55]. Traditional logic locking produces a locked circuit by mutating certain expressions based on input key bits. HOLL differs from logic locking on the type of entities employed as hidden keys. While logic locking uses a *key value* (i.e., a sequence of key-bits), our technique uses a *key relation* (i.e., a functional relation among the key bits). As we attempt to hide a higher-order entity (relation), we refer to our scheme as *higher-order logic locking* (HOLL). As synthesizing a relation (a second-order problem) is more challenging to recover than a bit-sequence (a first-order problem), HOLL is, at least in theory, is more secure than logic locking. Our experimental results (§6) show that this security also translates to practice.

Hardware constraints. Since the key relation must be implemented in the circuit, we need to consider practical constraints. For example, the size of the key relation affects the size of the programmable logic to be used for its implementation. This, in turn, introduces area and delay overheads in the final circuit. The practical realizability of this technique adds certain hardware constraints:

- The key relation must be small for it to have a small area overhead;
- The key relation must only be executed once to avoid a significant performance overhead;
- The key relation must encode non-trivial relations between the challenge and response bits to strong security;
- The locked expressions are evenly distributed across the design to protect all parts of the circuit, disallowing focused attacks by an attacker on a small part of the circuit that contains all locks.

Inferring the key relation. HOLL operates by

1. carefully selecting a set of expressions, $E \subseteq \varphi$, in the original design φ;
2. mutating each expression $e_i \in E$ using the relation bits R to create the corresponding *locked expression*, $\hat{e}_i$.

For example, in Fig. 1a, we select two expressions, $E = \{e_1, e_2\}$ where $e_1 = x_1 \wedge t_0$ and $e_2 = x_3 \wedge t_0$. e_1 computes t_2 and is a function of t_0 and x_1, while $\hat{e}_1$ uses x_0 and r_3, which is in turn a relation of r_0 and r_1. We formalize our lock program synthesis problem as follows.

Lock Inference. Given a circuit $Y \leftrightarrow \varphi(X)$, construct a *locked circuit* $Y \leftrightarrow \hat{\varphi}(X, R)$ and a *key relation* $\psi(X, R)$ such that $\hat{\varphi}$ is *semantically equivalent* to φ with the correct relation ψ. Specifically, it requires us to construct: (1) a key relation ψ and (2) a set of locked expressions $\hat{E}$ relating to the set of selected expressions E extracted from φ such that the following conditions are met:

- **Correctness**: The circuit is guaranteed to work correctly for all inputs when the key relation is installed:

$$\forall X. \ (\forall R. \ \psi(X, R) \implies (\hat{\varphi}(X, R) = \varphi(X))) \tag{1}$$

  where $\hat{\varphi} \equiv \varphi[\hat{e}_1/e_1, \ldots, \hat{e}_n/e_n]$, for $e_i \in E \subseteq \varphi$. The notation $\varphi[e_a/e_b]$ implies that e_b is replaced by e_a in the formula φ.
- **Security**: There must exist some relation ψ' (where $\psi' \neq \psi$) where the circuit exhibits incorrect behavior; in other words, it enforces the key relation to be non-trivial:

$$\exists \psi \ \exists X \ \exists R. \ (\psi'(X, R) \implies \hat{\varphi}(X, R) \neq \varphi(X)) \tag{2}$$

We pose the above as a *program synthesis* [40, 50] problem. In particular, we search for "mutations" $\hat{e}_1, \ldots, \hat{e}_2$ and a suitable key relation ψ such that the above constraints are satisfied.

2.3   Attacking with *SynthAttack*

As we attempt to hide a relation instead of a key-value, prior attacks on logic locking (like SAT attacks), which attempt to infer key bit-strings, do not apply. However, the attackers can also use program synthesis techniques to recover the key relation using an activated instance of the circuit as an input-output oracle.

We design an attack algorithm, called *SynthAttack*, combining ideas from SAT attack (for logic locking) and counterexample guided inductive program synthesis. Our attack algorithm generates inputs $X_1, X_2, \ldots, X_n$ and computes the corresponding outputs $Y_1, Y_2, \ldots, Y_n$ using the oracle, to construct a set of *examples* $\Lambda = \{(X_1, Y_1), \ldots, (X_n, Y_n)\}$. Then, the attacker can generate a key relation ψ that satisfies the above examples, λ, using a program synthesis query:

$$\prod_{X_i, Y_i \in \Lambda} \exists R_i. \ \hat{\varphi}(X_i, R_i) \wedge \psi(X_i, R_i) = Y_i \tag{3}$$

The above query requires *copies* of $\hat{\varphi}(X, R)$ for every example—hence, the formula will quickly explode with an increasing number of samples. Our scheme is robust since the sample complexity of the key relationships increases exponentially with the number of relation bits employed. Additionally, the attacker does not know which input bits excite the key relation and how the relation bits are related to each other.

For the locked adder (Fig. 1b) with the input samples shown in Table 1 (first four rows), the above attack can synthesize the key relation shown in Fig. 2. Columns Y and $\hat{Y}$ in Table 1 represent the outputs of the original circuit and the circuit obtained by the attacker, respectively. Even on a 4-bit input space, when 25% of all possible samples are provided, the attack fails to recover the key relation as shown by the last input row of Table 1. The red box highlights the output in the attacker circuit does not match the original design.

Table 1: In-out samples.

X	Y	$\hat{Y}$
1111	110	110
1001	011	011
0000	000	000
1100	011	011
0101	010	110

$\{(r_0 \leftrightarrow x_2),$
$(r_1 \leftrightarrow x_0),$
$(r_2 \leftrightarrow 0),$
$(r_3 \leftrightarrow r_0 \wedge r_1),$
$(r_4 \leftrightarrow 0)\}$

Fig. 2: Generated key relation

Definition (Distinguishing Input). Given a locked circuit $\hat{\varphi}$, we refer to input X as a distinguishing input if there exist candidate relations ψ_1 and ψ_2 that evoke semantically distinct behaviors on the input X. Formally, X is a distinguishing input provided the following formula is satisfiable[5] on some relations ψ_1 and ψ_2:

$$\hat{\varphi}(X, R_1) \neq \hat{\varphi}(X, R_2) \wedge \psi_1(X, R_1) \wedge \psi_2(X, R_2) \tag{4}$$

It searches for a *distinguishing input*, X_d, that produces conflicting outputs on the locked design. Any such distinguishing input is added to the set of examples, Λ, and the query repeated. If the query is unsatisfiable, it implies that no other relation can produce a different behavior on the locked design and so the relation that satisfies the current set of examples must be a correct key relation.

Though SynthAttack significantly reduces the sample complexity of the attack, our experiments demonstrate that SynthAttack is still unsuccessful at breaking HOLL for larger designs.

3 Program Synthesis to Infer Key Relations

We represent the key relation ψ as a propositional formula, represented as a set (conjunction) of propositional terms. The terms in ψ belong are categorised as:

- **Stimulus terms**: As mentioned in §2, a subset of the relation bits are related to input bits or constants; the stimulus terms appear as $(r_i \leftarrow x_j)$ where $r_i \in R, x_j \in X \cup \{0, 1\}$.
- **Latent terms**: These clauses establish a relation among the relation bits; the variables v in these terms are drawn from the relation bits R, i.e. $v \in R$.

---

[5] All free variables are existentially quantified.

For example, in Fig. 1c the terms $(r_0 \leftarrow x_1)$, $(r_1 \leftarrow x_2)$, and $(r_2 \leftarrow rand)$ are stimulus terms, while $(r_3 \leftarrow (r_0 \wedge r_1))$ and $(r_4 \leftarrow (r_1 \oplus r_2))$ are latent terms.

Budget. As the key relation may need to be implemented within a limited hardware budget, our synthesis imposes a hard threshold on its size. The threshold could directly capture the hardware constraints for implementing the key relation (e.g., the estimated number of gates or ports) or indirectly indicate the complexity of the key relation (e.g., number of relation bits, propositional terms, or latent terms).

3.1   Lock and Key Inference

Algorithm 1 outlines our algorithm for inferring the key relation and the locked circuit. The algorithm accepts an unlocked design $Y \leftrightarrow \varphi(X)$ and a budget T for the key relation.

Main Algorithm. The algorithm iterates, increasing the complexity of the key relation, till the budget T is reached (Lines 4-21). In every iteration, the algorithm selects a set of suitable expressions E for locking, uses our synthesis procedure to extract a set of additional latent terms H for the key relation, and produces the mutated expressions $\hat{e}_i$ for each expression $e_i \in E$ (Line 6). If the additional synthesized relations keep the key relation within the budget T (Line 8), the mutated expressions are replaced for $e_i \in E$ (Line 11). HOLL verifies that the solution meets

---

Algorithm 1: $\mathrm{HOLL}(\varphi, T, Q)$

```
1  ψ ← ∅
2  φ̂ ← φ
3  done ← False
4  while not done do
5  │   E ← SelectExpr(φ̂)
6  │   H, Ê ← Synthesize(ψ, φ̂, E)
7  │   ψ' ← ψ ∪ H
8  │   if Budget(ψ') ≤ T then
9  │   │   ψ ← ψ'
10 │   │   φ̂ ← φ̂[{êi/ei │
11 │   │         ei ∈ Ei, êi ∈ Ê}]
12 │   else
13 │   │   q ← CheckSec(ψ, φ̂)
14 │   │   if q then
15 │   │   │   done ← True
16 │   │   else
17 │   │   │   ψ ← ∅
18 │   │   │   φ̂ ← φ
19 │   │   end
20 │   end
21 end
22 return φ̂, ψ
```

---

the two objectives of correctness and security (§2). We handle correctness in the *Synthesize* procedure of Algorithm 1 and security in Lines 13-14 of the same algorithm. The *CheckSec()* procedure verifies if the synthesized (locked) circuit and key relations satisfy the security condition (Eqn 2). If *CheckSec()* returns True, the key relation ψ and the locked circuit $\hat{\varphi}$ are returned; otherwise, synthesis is reattempted.

Correctness. HOLL attempts to synthesize (via the *Synthesize* procedure) a key relation ψ and a set of locked expressions $\hat{e}_i$ such that the circuit is guaranteed to work correctly for all inputs given to ψ; this requires us to satisfy:

$$\exists \psi, \hat{e}_1, \ldots, \hat{e}_n. \ \forall X. \ \forall R. \ (\psi(X, R) \implies \hat{\varphi}(X, R) = \varphi(X)) \qquad (5)$$

$(r_0 \leftarrow x_1),$
$(r_1 \leftarrow x_2), (r_2 \leftarrow x_0),$
$(r_5 \leftarrow (r_0 \wedge r_1)),$
$(r_4 \leftarrow (\boxed{(r_0 \wedge r_1)} \wedge r_2)),$
$(r_3 \leftarrow (\boxed{(r_0 \wedge r_1)} \vee r_2))$

$(r_0 \leftarrow x_1),$
$(r_1 \leftarrow x_2),$
$(r_2 \leftarrow x_0),$
$(r_5 \leftarrow (r_0 \wedge r_1)),$
$(r_3 \leftarrow (\boxed{r_5} \vee r_2)),$
$(r_4 \leftarrow (\boxed{r_5} \wedge r_2))$
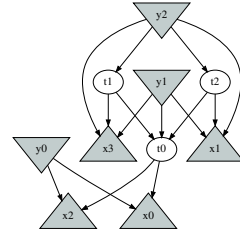
(a) Without optimization   (b) With opt.

Fig. 3: Key relations generated without and with optimization.

Fig. 4: Dependency graph for the expressions in Fig. 1a.

where $\hat{\varphi} \equiv \varphi[\hat{e}_1/e_1, \ldots, \hat{e}_n/e_n]$, for $e_i \in (E \subseteq \varphi)$. In other words, we attempt to synthesize a set of modified expressions $\hat{E}$ that, once replaced the selected expressions in E, produces a semantically equivalent circuit as the original circuit if the relation ψ holds.

We solve this synthesis problem via counterexample-guided inductive synthesis (CEGIS) [3]. We provide a domain-specific language (DSL) in which ψ and $\hat{e}_i$ are synthesized. CEGIS generates candidate solutions for the synthesis problem and uses violations to the specification (i.e. the above constraint) to guide the search for suitable *programs* ψ and $\hat{e}_i$.

A problem with the above formulation is illustrated in Fig. 3: the key relation in Fig. 3a uses 5 gates without reusing expressions, "wasting" hardware resources. Fig. 2b shows an optimized key relation that reuses the response bit r_5, allowing an implementation with only 3 gates. To encourage subexpression reuse, we solve the following optimization problem where $\hat{\varphi} \equiv \varphi[\hat{e}_1/e_1, \ldots, \hat{e}_n/e_n]$, for $e_i \in E \subseteq \varphi$.:

$$\underset{\mathsf{budget}(\psi)}{\mathsf{argmin}} \; \exists \psi, \hat{e}_1, \ldots, \hat{e}_n. \; \forall X. \; (\forall R. \; \psi(X, R) \implies \hat{\varphi}(X, R) = \varphi(X)) \quad (6)$$

Security. The security objective requires that the locking (i.e., the key relation ψ and the locked expressions) is non-trivial; that is. there exists some relation ψ':$\psi' \neq \psi$ for which the circuit is not semantically equivalent to the original design:

$$\exists \psi', \psi' \neq \psi, \; \text{s.t.} \; \exists X. \, (\exists R. \; \psi'(X, R) \wedge \hat{\varphi}(X, R) \neq \varphi(X)) \quad (7)$$

The above constraint is difficult to establish while synthesizing ψ; it requires a search for a different relation ψ' that makes $\hat{\varphi}$ semantically distinct from φ while ψ maintains semantic equivalence. Instead, we use a two-pronged strategy:

– We carefully design the DSL used to synthesize ψ and $\hat{e}_i$ to reduce the possibility they generate trivial relations;
– After obtaining ψ and $\hat{\varphi}$, we attempt to synthesize an alternative relation ψ' (using 8) such that $\hat{\varphi}$ is not semantically equivalent to φ, ensuring that ψ and $\hat{\varphi}$ do not constitute a trivial locked circuit.

$$\exists \psi'. \; \exists X, R'. \; \varphi(X) \neq \hat{\varphi}(X, R') \wedge \psi'(X, R') \qquad (8)$$

The procedure $CheckSec(\psi, \hat{\varphi})$ (Algorithm 1, Line 13) implements the above check (Eqn. 8).

Theorem 1. *If Algorithm 1 terminates, it returns a correct (Eqn. 1) and secure (Eqn. 2) locked design.*

Proof. The proof follows trivially from the design of the *Synthesize* (in particular, Eqn. 5) and *CheckSec* (in particular, Eqn. 8) procedures.

3.2 Expression Selection

HOLL constructs the dependency graph [19] (V, D) for expression selection, where nodes V are circuit variables. A node $v \in V$ represents an expression e such that v is assigned the result of expression e, i.e. $(v \leftarrow e)$. The edges D are dependencies: the edge $v_1 \rightarrow v_2$ connects the two nodes v_1 to v_2 if variable v_1 depends on variable v_2. The tree is rooted at the output variables and the input variables appear as leaves.

For example, Fig. 4 shows the dependency graph for the circuit in Fig. 1a. Triangles represent input ports (x_0, x_1, x_2, x_3) while inverted triangles represent output ports (y_0, y_1, y_2).

Our variable selection algorithm has the following goals:

1. **Ensure expression complexity**: The algorithm selects an expression e_z as a candidate for locking only if the depth of the corresponding variable z in the dependency graph lies in a user-defined range [L, U] to create a candidate set Z. The lower threshold L assures the expression captures a reasonably complex relation over the inputs, while the upper threshold U ensures the relation is not too complex to exceed the hardware budget. The algorithm starts by randomly selecting a variable $z_0 \in Z$ from this set.
2. **Encourage sub-expression reuse in key relation**: We attempt to select multiple "close" expressions; for the purpose, the algorithm randomly selects variables $w_i \subseteq Z$ on which z_0 (transitively) depend on.
3. **Encourage coverage**: We select expressions for locking in a manner so as to *cover* the circuit. To this end, interpreting (V, D) as an undirected graph, we randomly select expressions $u_i \in Z$ that are *farthest* from z_0, i.e. the shortest distance between u_i and z_0 is maximized.

Our algorithm first executes step (1), and then, indeterminately alternates between (2) and (3), till the required number of variables are selected. Let us use the dependency graph in Fig. 4 to show how the above algorithm operates:

- Given the user-defined range `[1,3]`, we compose the initial candidate set $Z = \{y_0, t_0, t_1, t_2, y_1, y_2\}$.
- Let us assume we randomly pick the expression for y_2. Now, y_2 depends on expressions t_0, t_1 and t_2 ($\{t_0, t_1, y_2\} \subseteq Z$) [*Rule 1*].

- We randomly choose new expressions to lock/transform from $\{t_0, t_1, y_2\}$. For example, we select t_2 and t_0—[*Rule 2*].
- We find y_0, which is the furthest expression from t_0, t_2, y_2 in Z. We select to lock the set of expressions $\{y_1, y_2, t_0, t_2\}$—[*Rule 3*].

4   HOLL: Implementation and Optimization

Implementation. We implemented HOLL in Python, using SKETCH [50] synthesis engine to solve the program synthesis queries. We used BERKELEY-ABC [8] to convert the benchmarks into Verilog and PyVERILOG [53], a Python-based library, to parse the Verilog code and generate input for SKETCH. We use the support for optimizing over a synthesis queries provided by SKETCH to solve Eqn. 6. Algorithm 1 may not terminate; our implementation uses a timeout to ensure termination.

Domain Specific Language. We specify our domain-specific language for synthesizing our key relations and locked expressions. The grammar is specified as generators in the SKETCH [50] language. The grammar for the key relations and locked expressions is as follows:

$$\langle G \rangle ::= r \leftarrow x \mid r \leftarrow r\langle Bop \rangle r \mid r \leftarrow \langle Uop \rangle r \mid r \leftarrow r$$
$$\langle Bop \rangle ::= or \mid and \mid xor$$
$$\langle Uop \rangle ::= not$$

The rule $\langle G \rangle ::= r \leftarrow x$ is only present in the key relation grammar since the locked expressions have no input bits.

Backslicing. To improve scalability, we use backslicing [56] to extract the portion of the design related to the expressions selected for locking. For a variable v_i, the set of all transitive dependencies that can affect the value of v_i is referred to as its *backslice*. For example, in Fig. 4, $backslice(t2) = \{t_0, x_0, x_1, x_2\}$.

   Given the set of expressions E, we compute the union of the backslices of the variables in E, i.e. all expressions B in the subgraph induced by $e \in E$ in the dependency graph; we use $B \subseteq E$ for lock synthesis.

   Backslicing tilts the asymmetrical advantage further towards the HOLL defense. The attacker cannot apply backslicing on the locked design since the dependencies are obscured, preventing the extraction of the dependency graph.

Incremental Synthesis. Given a set of expressions E, the procedure *Synthesis* in Algorithm 1 creates a list of relations H and a new set of locked expressions $\hat{E}$. If the list of expressions is large, we select the expressions in the increasing order of their depth in the dependency graph. The lower the depth of the expression is, the closer it is to the inputs, and the simpler is the expression. Selecting an expression with the lowest depth first (say e_1) ensures that other expressions (e_j)

depending on this expression can use the relations H generated during synthesis of $\hat{e}_1$. This also makes synthesizing the other expressions easier as the current relation has some sub-expressions on which the new relations can be built.

5   SynthAttack: Attacking HOLL with Program Synthesis

As HOLL requires inference of relations and not values, existing attacks designed for logic locking do not apply. We design a new attack, SynthAttack, that is inspired by the SAT attack [52] for logic locking and counterexample-guided inductive program synthesis (CEGIS) [51].

5.1   The SynthAttack Algorithm

SynthAttack runs a CEGIS loop: it accumulates a set of examples, Λ. These examples, Λ, are used to constrain the space of the candidate key-relations. SynthAttack, then, uses a verification check to confirm if the collected examples are sufficient to synthesize a valid key-relation. Otherwise, the counterexample from the failed verification check is identified as an `distinguishing input` (§2) to be added to Λ, and the algorithm repeats.

If there does not exist any distinguishing input for the locked circuit $\hat{\varphi}$, then $\hat{\varphi}$ has

---

Algorithm 2: SynthAttack($\hat{\varphi}$, $[\![\hat{\varphi}(\psi)]\!]$)

1 $i \leftarrow 0$
2 $Q_0 \leftarrow \top$
3 **while** *True* **do**
4 $X' \leftarrow Solve_X(Qi$
5 $\wedge\ (\hat{\varphi}(X, R_1) \neq \hat{\varphi}(X, R_2))$
6 $\wedge\ \psi_1(X, R_1) \wedge \psi_2(X, R_2))$
7   **if** $X' = \bot$ **then**
8   │   **break**
9   **end**
10 $Y' \leftarrow [\![\hat{\varphi}(\psi)]\!](X')$
11 $Q_{i+1} \leftarrow Q_i\ \wedge\ (\hat{\varphi}(X', R_1^i) \leftrightarrow Y')$
12 $\wedge\ (\hat{\varphi}(X', R_2^i) \leftrightarrow Y')$
13 $\wedge\ \psi_1(X', R_1^i)\ \wedge\ \psi_2(X', R_2^i)$
14 $i \leftarrow i + 1$
15 **end**
16 $\psi_1, \psi_2 \leftarrow Solve_{\psi_1, \psi_2}(Q_i)$
17 **return** ψ_1

---

a unique semantic behavior—and that must be the correct functionality. Any key-relation that satisfies the counterexamples (distinguishing inputs) generated so far will be a valid candidate for the key relation. An inductive synthesis strategy based on distinguishing inputs allows us to quickly converge on a valid realization of the key-relation as each distinguishing input disqualifies many potential candidates for the key relation. Note that (as we illustrate the following example) there may be multiple, possibly semantically dissimilar, realizations of a key-relation that enables the same (correct) functionality on the locked circuit.

SynthAttack is outlined in Algorithm 2: the algorithm accepts the design of the locked circuit ($\hat{\varphi}$) and an activated circuit $\hat{\varphi}(\psi)$ (the locked circuit $\hat{\varphi}$ activated with a valid key-relation ψ). The notation $[\![\hat{\varphi}(\psi)]\!]$ indicates that this activated circuit can only be used as an input-output oracle, but cannot be inspected.

SynthAttack runs a counterexample-guided synthesis loop (Line 3). It checks for the existence of a distinguishing input in Line 4: if no such distinguishing input exists, it implies that the current set of examples is sufficient to synthesize a valid key-relation. So, in this case, the algorithm breaks out of the loop (Line 7-8) and synthesizes a key-relation (Line 16), that is returned as the synthesized, provably-correct instance for the key relation.

If there exists a distinguishing input X' (in Line 4), the algorithm uses the activated circuit to compute the expected output Y' corresponding to X (Line 10). This new counterexample (X', Y') is used to *block* all candidate key-relations that lead to an incorrect behavior, thereby reducing the potential choices for ψ_1 and ψ_2. The loop continues, again checking for the existence of distinguishing inputs on the updated constraint for Q_i.

The theoretical analysis of SynthAttack is available in the extended version [54]. The algorithm only terminates when it is able to synthesize a provably valid key-relation, that allows us to state the following result.

Theorem 2. *Algorithm 2 will always terminate, returning a key-relation ψ_1 such that $\hat{\varphi}(\psi_1)$ is semantically equivalent to $\hat{\varphi}(\psi)$, where ψ is the "correct" relation hidden by HOLL. (The proof is available in the extended version [54].)*

Example. SynthAttack on Fig. 1b iteratively generates six distinguishing inputs (Table 2). The key relation synthesized by SynthAttack (Fig. 5) is **not** semantically equivalent to the hidden key-relation that was computed and hidden by HOLL (Fig. 1c). This shows that there may exist multiple valid candidates for the key-relation that all evoke the same functionality on the locked design. For example, $X = 0100$ generates $r_4 = 1$ for the key relation in Fig. 1c but $r_4 = 0$ for Fig. 5; however, the output of the locked circuit remains the same in both cases ($Y = 001$).

$$\{(r_0 \leftrightarrow x_1),$$
$$(r_1 \leftrightarrow x_2),$$
$$(r_2 \leftrightarrow r_0),$$
$$(r_3 \leftrightarrow r_2 \wedge r_1),$$
$$(r_4 \leftrightarrow \neg r_2 \wedge r_1)\}$$

Fig. 5: Key relation generated by SynthAttack.

Table 2: Distinguishing inputs.

X	Y
1101	100
0001	001
0101	010
0111	100
1001	011
0011	011

6   Experimental Evaluation

We selected 100 combinational benchmarks from ISCAS'85 [1] and MCNC [59] and report the time for program synthesis and the overhead after applying our locking method. For long running experiments, we select a subset of 10 randomly selected benchmarks where, number of input ports range between 16 and 256, output ports range range between 7 and 245, AND gates in range [135, 4174].

For our experiments, we use number of relation terms as *budget* in the range [12-14] for the key relation and depth of expression selection in range [2-4]. We conduct our experiments on a machine with 32-Core Intel(R) Xeon(R) Silver 4108 CPU @ 1.80GHz with 32GB RAM.

For both HOLL and SynthAttack, we use the SKETCH synthesis tool. Since synthesis solvers are difficult to compare across different problem instances, we

were wary of the case where the defender gets an edge over the attacker due to use of different tools. We create the attack-team-defence-team asymmetry by controlling the computation time: while the defender gets 20 minutes (1200s) to generate locked circuit, the attacker runs the attack for up to 4 days.

Our experiments aim to answer five research questions:

RQ1. What is the attack resilience of HOLL? (§6.1)

RQ2. How do impact expression selection heuristics affect attack resilience? (§6.2)

RQ3. What is the hardware cost for HOLL? (§6.3)

RQ4. What is the time taken to synthesize the locked design and key-relation for HOLL? (refer to the extended version [54])

RQ5. What are the impact of the optimizations for scalability (backslicing and incremental synthesis)? (refer to the extended version [54])

Here is a summary of our findings:

> **Security.** The key relations can be recovered completely by the attacker via SynthAttack but only for small circuits with a small hardware budget. For medium and large designs, key relations are fast to obtain ($<$1200s) but cannot be recovered by our attack even within 4 days. This shows our defense is efficient while our attack is strong but not scalable.
>
> **Hardware Cost.** Our key relations with a budget of 12-14 latent terms have a minimal impact on the designs and the overhead reduces as the size of the circuit grows. On the largest benchmark, the area overhead is 1.2%. The corresponding configurations for programmable devices are small and provide high security.
>
> **HOLL Performance.** The HOLL execution time ranges between 8s and 1001s, with an average of 33s for small, 17s for medium, and 60s for large designs for the budget of 8-10 latent terms. Our optimizations are crucial for the scalability of our HOLL defense (locking) algorithm: we fail to lock enough expressions in large circuits without these optimizations.

6.1   Attack Resilience

We define *attack resilience* of locked circuit, $\hat{\varphi}$, in terms of time taken to obtain a key relation, ψ', such that $\hat{\varphi} \wedge \psi'$ is equivalent to original circuits, φ.

Attack time. Fig. 6 shows the cumulative time spent till the ith iteration (y-axis) of the loop versus the loop counter i, that is also the number of distinguishing inputs (samples) generated so far (x-axis). We show exponential *trend* curves (as a solid pink line) to capture the trend in the plotted points while the data-points are plotted as blue dots. The plots show that the plotted points follow the exponential trend lines, illustrating that SynthAttack does not scale well, thereby asserting the resilience of HOLL.

SynthAttack failed to construct a valid key-relation for any of these ten designs within a timeout of 4 days. However, for small designs with lesser number of latent terms, SynthAttack was able to construct a valid key-relation (Fig. 7).
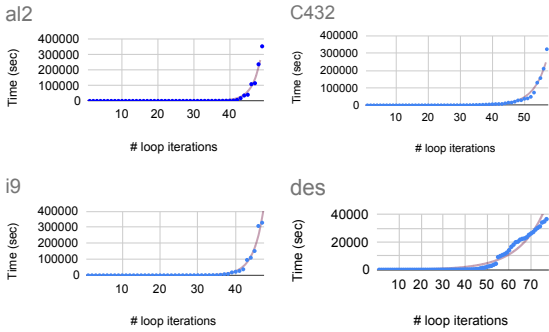
Fig. 6: Cumulative time for successive iterations of SynthAttack (best viewed in color)
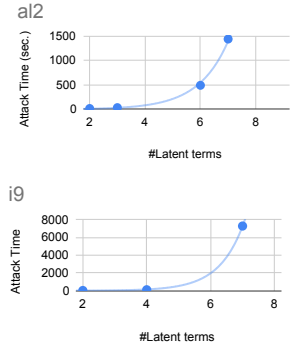
Fig. 7: Attack time vs #latent terms for i9 and al2

Attack resilience vs. number of latent terms. The complexity of the key relation increases with the number of relation bits. As shown in Fig. 7 (for benchmarks al2 and i9), the time required to break the locked circuit increases exponentially as the number of relation bits increases. We gave a timeout of 10 hours for this experiment and al2 timed out at 9 latent terms, and i9 timed out at 8 latent terms. Both results are for locked circuits with variables selected with the depth of locked expression, $\hat{e}_i$, equal to 1.

6.2   Impact of Expression Selection on Attack Resilience

Attack resilience vs. Depth of locked expression. The attack resiliency of $\hat{\varphi}$ increases significantly as we increase the depth of the locked expression selected for HOLL for φ. We observe that for a number of latent terms in key relation equal to 2, for benchmark al2, increases from 213s to 3788s for depth 1 and 2, respectively. For benchmark i9, attack time increases from 351s to 1141s for depth 1 and 2, respectively.

Attack time vs. Coverage. To show the effect of coverage we select expressions (in $e_i \in E$) such that the distance (§3.2) among the expressions is largest (termed as diverse) and smallest (termed as converged). The attack time to break the locked circuit is more for diverse than converged expression selection heuristic. For example, for benchmarks C432 and i9, attack time increases from 115s to 142s and 229s to 316s, respectively, when expression selection heuristic is changed from converged to diverse. The results are with three latent terms.

6.3   Hardware cost

The key relations can be implemented either as embedded Field Programmable Gate Array (eFPGA) or Programmable Array Logic. We synthesize the original and locked designs with SYNOPSYS DESIGN COMPILER R-2020.09-SP1 targeting the Nangate 15nm ASIC technology at standard operating conditions (25°C).

Table 3 provides the estimated cost for implementing the key relations with programmable devices. To do so, we compute the number of equivalent NAND2 gates used to estimate the number of 6-input LUTs. Given the number of LUTs, we give an estimation of the equivalent number of configuration bits (see [54] for details)–including those for switch elements. Results show that the size of the key relations is independent of original design size. Table 3 reports the fraction of the area locked with HOLL (*key relation*) to the area of the *original* circuit. The results show that the impact of HOLL is low, mainly for large designs.

Table 3: Hardware Impact of HOLL.

Bench	Orig. Area (μm^2)	Key Relation Area (μm^2)	#Eq. LUT	#Eq. conf. bits	Over-head Area (%)
al2	17.89	4.473	138	8,832	25.0
cht	20.74	4.178	132	8,448	20.1
C432	20.05	4.866	150	9,600	24.3
C880	50.04	4.325	132	8,448	8.6
i9	77.07	4.129	126	8,064	5.4
i7	80.41	4.129	126	8,064	4.3
x3	95.21	5.014	156	9,984	5.3
frg2	100.81	4.669	144	9,216	4.6
i8	120.37	4.325	132	8,448	3.6
des	445.37	5.554	174	11,136	1.2

7 Related Work

Logic Locking: Attacks and Defenses. Existing logic locking methods aptly operate on the gate-level netlists [55]. Gate-level locking cannot obfuscate all the semantic information because logic synthesis and optimizations absorb many of them into the netlist before the locking step. For example, constant propagation absorbs the constants into the netlist. Recently, alternative high-level locking methods obfuscate the semantic information before logic optimizations embed them into the netlist [37, 17]. For example, TAO applies obfuscations during HLS [37] but requires access to the HLS source code to integrate the obfuscations and cannot obfuscate existing IPs. Protecting a design at the register-transfer level (RTL) is an interesting compromise [29, 10]. Most of the semantic information (e.g., constants, operations, and control flows) is still present in the RTL and obfuscations can be applied to existing RTL IPs. In [29], the authors propose structural and functional obfuscation for DSP circuits. In [10], the authors propose a method to insert a special finite state machine to control the transition between obfuscated mode (incorrect function) and normal mode (correct function). Such transitions can only happen with a specific input sequence. Differently from [13], we extract the relation directly from the analysis of a single RTL design, making the approach independent of the design flow. None of these methods consider the possibility of hiding a relation among the key bits.

Program Synthesis. Program synthesis has been successful in many domains: synthesis of heap manipulations [39, 20, 58], bit-manipulating programs [27],

bug synthesis [41], parser synthesis [30, 47], regression-free repairs [6, 5], synchronization in concurrent programs [57], boolean functions [22, 24, 23] and even differentially private mechanisms [38]. There has also been an interest in using program synthesis in hardware designs [16]. VeriSketch [4] exploits the power of program synthesis in hardware design. Our work is orthogonal to the objectives and techniques of VeriSketch: while VeriSketch secures hardware against timing attacks, we propose a hardware locking mechanism. Zhang et al. [63] use SyGUS based program synthesis to infer environmental invariants for verifying hardware circuits. We believe that this work shows the potential of applying programming languages techniques in hardware design. We believe that there is also a potential of applying program analysis techniques, symbolic [9, 21, 12, 36, 34], dynamic [42, 14] and statistical [28, 32, 11, 33], for hardware analysis; this is a direction we intend to pursue in the future.

8   Discussion

We end the paper with an important clarification: the *eFPGA configuration* in HOLL can also be represented as a bit sequence (i.e., a sequence of *configuration* bits). So, why can an attacker not launch attacks similar to SAT attacks on logic locking to recover the HOLL configuration bitstream?

The foremost reason is that while the key-bits in traditional logic locking simply represent a *value* that the attacker attempts to recover, the bit-sequence in HOLL is an encoding of a *program* [15, 35]. This raw bit-sequence used to program an eFPGA is too "low-level" to be synthesized directly—the size of such bit-streams is about **60-85** times of the keys used in traditional logic locking (128 key bit-sequence). So, the HOLL algorithm designer uses a higher-level domain-specific language (DSL) to synthesize the key relation (see §4), that is later "compiled" to the configuration sequence. The attacker will also have to use a similar strategy of using a high-level DSL to break HOLL.

However, while the designer of the key relation can use a well-designed small domain-specific language (DSL) that includes the exact set of components required (and a controlled budget) to synthesize the key relation, the attacker, not aware of the key relation or the DSL, will have to launch the attack with a "guess" of a large overapproximation. In other words, **the domain-specific language used for synthesis is also a** *secret*, thereby making HOLL much harder to crack than traditional logic locking.

We evaluate HOLL (§6.1) under the assumption that the DSL (and budget) are known to the attacker. In real deployments (when the DSL is not known to the attacker), HOLL will be still more difficult to crack.

Bibliography

[1] ISCAS'85 benchmarks. https://filebox.ece.vt.edu/~mhsiao/iscas85.html, accessed: 2021-01-10

[2] Where is the eFPGA market and ecosystem headed? https://semiengineering.com/where-is-the-efpga-market-and-ecosystem-\headed/, accessed: 2021-05-28

[3] Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M.M.K., Raghothaman, M., Saha, S., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Irlbeck, M., Peled, D.A., Pretschner, A. (eds.) Dependable Software Systems Engineering, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 40, pp. 1–25. IOS Press (2015). https://doi.org/10.3233/978-1-61499-495-4-1

[4] Ardeshiricham, A., Takashima, Y., Gao, S., Kastner, R.: Verisketch: Synthesizing secure hardware designs with timing-sensitive information flow properties. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 1623–1638. CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3354246

[5] Bavishi, R., Pandey, A., Roy, S.: Regression aware debugging for mobile applications. In: Mobile! 2016: Proceedings of the 1st International Workshop on Mobile Development (Invited Paper). p. 21–22. Mobile! 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/3001854.3001860

[6] Bavishi, R., Pandey, A., Roy, S.: To be precise: Regression aware debugging. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2016, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2983990.2984014

[7] Bhandari, J., Moosa, A.K.T., Tan, B., Pilato, C., Gore, G., Tang, X., Temple, S., Gaillardon, P.E., Karri, R.: Exploring eFPGA-based Redaction for IP Protection. In: International Conference on Computer-Aided Design (ICCAD) (Nov 2021)

[8] Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. pp. 24–40. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

[9] Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. p. 209–224. OSDI'08, USENIX Association, USA (2008)

[10] Chakraborty, R.S., Bhunia, S.: RTL hardware IP protection using key-based control and data flow obfuscation. In: Proceedings of the International Conference on VLSI Design. pp. 405–410 (2010)

[11] Chatterjee, P., Chatterjee, A., Campos, J., Abreu, R., Roy, S.: Diagnosing software faults using multiverse analysis. In: Bessiere, C. (ed.) Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20. pp. 1629–1635. International Joint Conferences on Artificial Intelligence Organization (7 2020). https://doi.org/10.24963/ijcai.2020/226, main track

[12] Chatterjee, P., Roy, S., Diep, B.P., Lal, A.: Distributed bounded model checking. In: FMCAD (July 2020)

[13] Chen, J., Zaman, M., Makris, Y., Blanton, R.D.S., Mitra, S., Schafer, B.C.: DECOY: DEflection-Driven HLS-Based Computation Partitioning for Obfuscating Intellectual Property. In: Design Automation Conference (DAC). pp. 1–6 (2020)

[14] Chouhan, R., Roy, S., Baswana, S.: Pertinent path profiling: Tracking interactions among relevant statements. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 1–12 (2013). https://doi.org/10.1109/CGO.2013.6494983

[15] Clift, J., Murfet, D.: Encodings of turing machines in linear logic. Mathematical Structures in Computer Science **30**(4), 379–415 (2020)

[16] Cook, B., Gupta, A., Magill, S., Rybalchenko, A., Simsa, J., Singh, S., Vafeiadis, V.: Finding heap-bounds for hardware synthesis. In: 2009 Formal Methods in Computer-Aided Design. pp. 205–212 (2009). https://doi.org/10.1109/FMCAD.2009.5351120

[17] Di Crescenzo, G., Sengupta, A., Sinanoglu, O., Yasin, M.: Logic locking of boolean circuits: Provable hardware-based obfuscation from a tamper-proof memory. In: Simion, E., Géraud-Stewart, R. (eds.) Innovative Security Solutions for Information Technology and Communications. pp. 172–192. Springer International Publishing, Cham (2020)

[18] El Massad, M., Garg, S., Tripunitara, M.: Integrated circuit (ic) decamouflaging: Reverse engineering camouflaged ics within minutes. In: Network and Distributed System Security Symposium (NDSS) (01 2015). https://doi.org/10.14722/ndss.2015.23218

[19] Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (Jul 1987). https://doi.org/10.1145/24039.24041

[20] Garg, A., Roy, S.: Synthesizing heap manipulations via integer linear programming. In: Blazy, S., Jensen, T. (eds.) Static Analysis, SAS 2015. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48288-9_7

[21] Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1065010.1065036

[22] Golia, P., Roy, S., Meel, K.S.: Manthan: A data-driven approach for boolean function synthesis. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification (CAV). pp. 611–633. Springer International Publishing, Cham (2020)

[23] Golia, P., Roy, S., Meel, K.S.: Program synthesis as dependency quantified formula modulo theory. In: Zhou, Z.H. (ed.) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21. pp. 1894–1900. International Joint Conferences on Artificial Intelligence Organization (8 2021). https://doi.org/10.24963/ijcai.2021/261, main Track

[24] Golia, P., Roy, S., Slivovsky, F., Meel, K.S.: Engineering an efficient boolean functional synthesis engine. In: ICCAD (2021)

[25] Guin, U., Huang, K., DiMase, D., Carulli, J.M., Tehranipoor, M., Makris, Y.: Counterfeit Integrated Circuits: A rising threat in the global semiconductor supply chain. Proceedings of the IEEE **102**(8), 1207–1228 (Aug 2014)

[26] Hurtarte, J., Wolsheimer, E., Tafoya, L.: Understanding Fabless IC Technology. Elsevier (Aug 2007)

[27] Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1. ICSE '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1806799.1806833

[28] Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of Test Information to Assist Fault Localization. In: Proceedings of the 24th International Conference on Software Engineering. ICSE '02, ACM, New York, NY, USA (2002). https://doi.org/10.1145/581339.581397

[29] Lao, Y., Parhi, K.K.: Obfuscating DSP circuits via high-level transformations. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **23**(5), 819–830 (2015)

[30] Leung, A., Sarracino, J., Lerner, S.: Interactive parser synthesis by example. SIGPLAN Not. **50**(6), 565–574 (Jun 2015). https://doi.org/10.1145/2813885.2738002

[31] Li, L., Orailoglu, A.: Piercing logic locking keys through redundancy identification. In: Design, Automation and Test in Europe Conference (DATE). pp. 540–545 (2019)

[32] Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable Statistical Bug Isolation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1065010.1065014

[33] Modi, V., Roy, S., Aggarwal, S.K.: Exploring Program Phases for Statistical Bug Localization. In: Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. PASTE '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2462029.2462034

[34] Pandey, A., Kotcharlakota, P.R.G., Roy, S.: Deferred concretization in symbolic execution via fuzzing. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 228–238. ISSTA 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293882.3330554

[35] Petersen, H.: Some remarks on real-time turing machines. arXiv preprint arXiv:1902.00975 (2019)

[36] Pham, V.T., Khurana, S., Roy, S., Roychoudhury, A.: Bucketing failing tests via symbolic analysis. In: Huisman, M., Rubin, J. (eds.) Fundamental Approaches to Software Engineering. pp. 43–59. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)

[37] Pilato, C., Regazzoni, F., Karri, R., Garg, S.: TAO: Techniques for algorithm-level obfuscation during high-level synthesis. In: Design Automation Conference (DAC). pp. 1–6 (Jun 2018)

[38] Roy, S., Hsu, J., Albarghouthi, A.: Learning differentially private mechanisms. In: 2021 2021 IEEE Symposium on Security and Privacy (SP). pp. 852–865. IEEE Computer Society, Los Alamitos, CA, USA (May 2021). https://doi.org/10.1109/SP40001.2021.00060

[39] Roy, S.: From concrete examples to heap manipulating programs. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_9

[40] Roy, S., Pandey, A., Dolan-Gavitt, B., Hu, Y.: Bug synthesis: Challenging bug-finding tools with deep faults. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 224–234. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018)

[41] Roy, S., Pandey, A., Dolan-Gavitt, B., Hu, Y.: Bug synthesis: Challenging bug-finding tools with deep faults. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 224–234. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3236024.3236084

[42] Roy, S., Srikant, Y.N.: Profiling k-iteration paths: A generalization of the ball-larus profiling algorithm. p. 70–80. CGO '09, IEEE Computer Society, USA (2009). https://doi.org/10.1109/CGO.2009.11

[43] S. W. Jones: Technology and Cost Trends at Advanced Nodes. IC Knowledge LLC (2019)

[44] Shamsi, K., Li, M., Meade, T., Zhao, Z., Pan, D.Z., Jin, Y.: Circuit obfuscation and oracle-guided attacks: Who can prevail? In: Great Lakes Symposium on VLSI. pp. 357–362. ACM, New York, NY, USA (2017)

[45] Shamsi, K., Li, M., Plaks, K., Fazzari, S., Pan, D.Z., Jin, Y.: IP protection and supply chain security through logic obfuscation: A systematic overview. ACM Transactions on Design Automation of Electronic Systems **24**(6) (Sep 2019)

[46] Shamsi, K., Pan, D.Z., Jin, Y.: On the impossibility of approximation-resilient circuit locking. In: IEEE International Symposium on Hardware Oriented Security and Trust. pp. 161–170 (2019)

[47] Singal, D., Agarwal, P., Jhunjhunwala, S., Roy, S.: Parse condition: Symbolic encoding of ll(1) parsing. In: Barthe, G., Sutcliffe, G., Veanes, M. (eds.) LPAR-22. 22nd International Conference on Logic for Programming,

Artificial Intelligence and Reasoning. EPiC Series in Computing, vol. 57, pp. 637–655. EasyChair (2018). https://doi.org/10.29007/2ndp

[48] Sirone, D., Subramanyan, P.: Functional analysis attacks on logic locking. In: Design, Automation & Test Conference in Europe (DATE). pp. 1–6 (Mar 2019)

[49] Sisejkovic, D., Merchant, F., Reimann, L.M., Srivastava, H., Hallawa, A., Leupers, R.: Challenging the security of logic locking schemes in the era of deep learning: A neuroevolutionary approach (2020)

[50] Solar-Lezama, A.: The sketching approach to program synthesis. In: Hu, Z. (ed.) Proceedings of Asian Symposium Programming Languages and Systems, 7th, (APLAS). vol. 5904, pp. 4–13. Springer (2009)

[51] Solar-Lezama, A.: Program sketching. vol. 15, p. 475–495. Springer-Verlag, Berlin, Heidelberg (Oct 2013). https://doi.org/10.1007/s10009-012-0249-7

[52] Subramanyan, P., Ray, S., Malik, S.: Evaluating the security of logic encryption algorithms. In: IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 137–143 (2015)

[53] Takamaeda-Yamazaki, S.: Pyverilog: A Python-based hardware design processing toolkit for Verilog HDL. In: arc. pp. 451–460 (Apr 2015)

[54] Takhar, G., Karri, R., Pilato, C., Roy, S.: HOLL: Program synthesis for higher order logic locking (2022), https://arxiv.org/abs/2201.10531

[55] Tan, B., Karri, R., Limaye, N., Sengupta, A., Sinanoglu, O., Rahman, M.M., Bhunia, S., Duvalsaint, D., Blanton, R., Rezaei, A., Shen, Y., Zhou, H., Li, L., Orailoglu, A., Han, Z., Benedetti, A., Brignone, L., Yasin, M., Rajendran, J., Zuzak, M., Srivastava, A., Guin, U., Karfa, C., Basu, K., Menon, V.V., French, M., Song, P., Stellari, F., Nam, G.J., Gadfort, P., Althoff, A., Tostenrude, J., Fazzari, S., Breckenfeld, E., Plaks, K.: Benchmarking at the frontier of hardware security: Lessons from logic locking. arXiv, https://arxiv.org/abs/2006.06806 (2020)

[56] Venkatesh, G.A.: The semantic approach to program slicing. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation. p. 107–119. PLDI '91, Association for Computing Machinery, New York, NY, USA (1991). https://doi.org/10.1145/113445.113455

[57] Verma, A., Kalita, P.K., Pandey, A., Roy, S.: Interactive debugging of concurrent programs under relaxed memory models. In: Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization. p. 68–80. CGO 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3368826.3377910

[58] Verma, S., Roy, S.: Synergistic debug-repair of heap manipulations. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3106237.3106263

[59] Yang, S.: Logic synthesis and optimization benchmarks user guide: Version 3.0. Tech. rep., MCNC Technical Report (Jan 1991)

[60] Yasin, M., Mazumdar, B., Sinanoglu, O., Rajendran, J.: Removal attacks on logic locking and camouflaging techniques. IEEE Transactions on Emerging Topics in Computing **8**(2), 517–532 (2020)

[61] Yasin, M., Sengupta, A., Nabeel, M.T., Ashraf, M., Rajendran, J.J., Sinanoglu, O.: Provably-secure logic locking: From theory to practice. In: Conference on Computer and Communications Security. pp. 1601–1618 (2017)
[62] Yasin, M., Sengupta, A., Schafer, B.C., Makris, Y., Sinanoglu, O., Rajendran, J.: What to lock? functional and parametric locking. In: Proceedings of the on Great Lakes Symposium on VLSI 2017. pp. 351–356 (2017)
[63] Zhang, H., Yang, W., Fedyukovich, G., Gupta, A., Malik, S.: Synthesizing environment invariants for modular hardware verification. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **11990 LNCS**, 202–225 (2020). https://doi.org/10.1007/978-3-030-39322-9_10