arXiv:2010.05344v1 [cs.CR] 11 Oct 2020

ASSURE: RTL Locking Against an Untrusted Foundry

Christian Pilato, *Senior Member, IEEE,* Animesh Basak Chowdhury,
Donatella Sciuto, *Fellow, IEEE,* Siddharth Garg, *Member, IEEE,* Ramesh Karri, *Fellow, IEEE*

***Abstract*—Semiconductor design companies are integrating proprietary intellectual property (IP) blocks to build custom integrated circuits (IC) and fabricate them in a third-party foundry. Unauthorized IC copies cost these companies billions of dollars annually. While several methods have been proposed for hardware IP obfuscation, they operate on the gate-level netlist, i.e., after the synthesis tools embed the semantic information into the netlist. We propose ASSURE to protect hardware IP modules operating on the register-transfer level (RTL) description. The RTL approach has three advantages: (i) it allows designers to obfuscate IP cores generated with many different methods (e.g., hardware generators, high-level synthesis tools, and pre-existing IPs). (ii) it obfuscates the semantics of an IC before logic synthesis; (iii) it does not require modifications to EDA flows. We perform a cost and security assessment of ASSURE.**

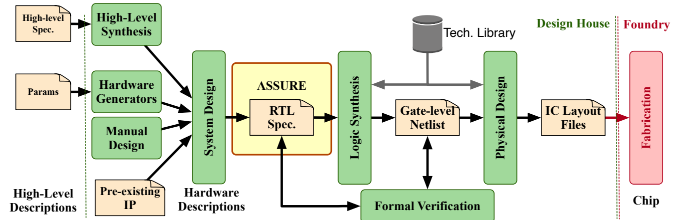


Fig. 1: State-of-the-art IC design flow. Designers create RTL description of an IC either by manual design, or by using HLS tools, or by using hardware generators. The netlist after more processing steps is sent to a third-party foundry. ASSURE locks an RTL description before logic synthesis.

I. Introduction

The cost of IC manufacturing has increased 5× when scaling from 90nm to 7nm [1]. An increasing number of design houses are now *fab-less* and outsource the fabrication to a third-party foundry [2], [3]. This reduces the cost of operating expensive foundries but raises security issues. If a rogue in the third-party foundry has access to the design files, they can reverse engineer the IC functionality to steal the Intellectual Property (IP), causing economic harm to the design house [4].

Fig. 1 is a fabless IC design flow with third-party manufacturing. The flow accepts the specification in a *hardware description language* (HDL). Designers create the components either manually or generate them automatically, and integrate them into a hardware description at the register-transfer level (RTL). Given a technology library (i.e., a description of gates in the target technology) and a set of constraints, logic synthesis elaborates the RTL into a gate-level netlist. Logic synthesis applies optimizations to reduce area and improve timing. While RTL descriptions are hard to match against high-level specifications [5], they are used as a golden reference during synthesis to verify each step does not introduce any error. Physical design generates the layout files that are sent to the foundry for fabrication of ICs that are then returned to the design house for packaging and testing.

Semiconductor companies are developing methods for IP obfuscation. In *split manufacturing*, the design house splits the IC into parts that are fabricated by different foundries [6]. An attacker must access all parts to recover the IC. *Watermarking* hides a signature inside the circuit, which is later verified during litigation [7]. Finally, designers apply *logic locking* [8] to prevent unauthorized copying and thwart reverse-engineering. They introduce extra gates controlled by a key that is kept secret from the foundry. They *activate* the IC functionality by installing the key into a tamper-proof memory after fabrication.

A. Related Work

Prior approaches lock gate-level netlists after logic optimizations have been applied [9]. Gate-level locking can not 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. When the attackers have access to an activated IC (i.e. the oracle), they use Boolean Satisfiability (SAT)-based attacks to recover the key [10], [11]. Several solutions have been proposed to thwart SAT-based attacks [12], [13]. Attacks on SFLL have been reported when the "protected" functional inputs are at a certain (Hamming) distance from the key [14], [15].

Recently, alternative high-level locking methods have been proposed [16], [17], [18], [19]. These methods obfuscate the semantic information before logic optimizations embed them into the netlist. TAO applies obfuscations during HLS [16]. HLS-based SFLL obfuscation has been proposed in [17]. Both approaches require access to the HLS source code to integrate the obfuscations and cannot be used to obfuscate existing IPs.

Protecting a design at the register-transfer level (RTL) is a compromise that ASSURE takes. Most of the semantic information (e.g., constants, operations and control flows) is

Manuscript received October 11, 2020.

C. Pilato and D. Sciuto are with the Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy (christian.pilato@polimi.it, donatella.sciuto@polimi.it).

A. B. Chowdhury, S. Garg and R. Karri are with the NYU Center for Cybersecurity (http://cyber.nyu.edu), New York University, New York, NY, USA (abc586@nyu.edu, garg@nyu.edu, rkarri@nyu.edu).

still present in the RTL and obfuscations can be applied to existing RTL IPs. To obfuscate the semantic information, ASSURE leverages prior work on software program obfuscation [20], [21], [22]. They obfuscate data structures, control flows and constants through code transformations or by loading information from memory at runtime.

B. Paper Contributions

ASSURE RTL obfuscation uses three techniques namely obfuscating constants, arithmetic operations, and control branches. These are provably secure and compatible with industrial design flows. The paper makes three contributions:

1) an RTL-to-RTL translation for IP obfuscation (Section III).
2) three obfuscations (constant, operations, and branch) with proofs of security (Section III-B).
3) reports on security (Section IV-B) and related overhead (Section IV-C).

II. Threat Model: Untrusted Foundry

The state-of-art in logic locking considers two broad categories of threat models: netlist-only and oracle-guided [8], [23]. In both settings, the attacker has access to a locked netlist, but in the latter, also has access to an unlocked IC (*oracle*). The oracle-guided model is relevant in high-volume commercial fabrication where it is reasonable to assume that the attacker can purchase an unlocked IC in the market. The nelist-only model, one the other hand, captures low-volume settings, for instance, in the design of future defense systems with unique hardware requirements [24], where the attacker would not reasonably be able to access a working copy of the IC. For this reason, it is also called *oracle-less* model. In this work, we consider the oracle-less model.

Consider, for instance, a fabless defense contractor that outsources an IC to an **untrusted foundry** for fabrication. The untrusted foundry has access to the layout files of the design and can reverse engineer a netlist and even extract the corresponding RTL [25]. However, since the foundry produces the first ever batch of an IC design (in some cases the only one), an activated chip is not available through any other means. Attacks that rely on knowledge of an IC's true I/O behaviour, for instance the SAT attack, are therefore out-of-scope. However, the attacker can still rely on a range of netlist-only attacks, *desynthesis* [26], *redundancy identification* [27] and *ML-guided structural and functional analysis* [28], [29], for instance, to reverse engineer the locked netlist. In the following, we prove the resilience of ASSURE obfuscation to not only these three attacks, but also that ASSURE locked netlists reveal *no information* about the design other than any prior knowledge that the designer might have about the design.

III. Overview of ASSURE

Fig. 2 shows the ASSURE flow. Given an RTL design D and a set of obfuscation parameters, ASSURE generates a design D^* and a key $\mathcal{K}_r^*$ such that D^* matches the functionality of D only when $\mathcal{K}_r^*$ is applied. ASSURE is a technology-independent and operates on the RTL after system integration but before logic synthesis. ASSURE obfuscates existing IPs

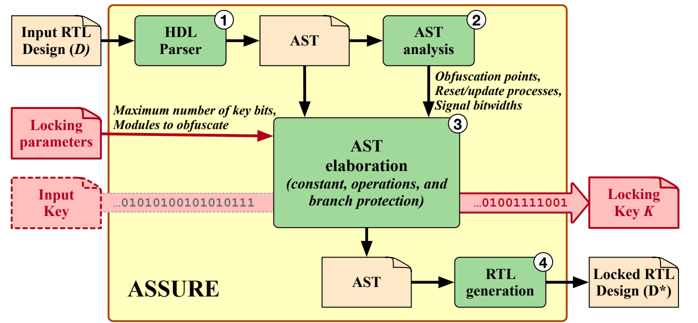


Fig. 2: Organization of ASSURE.

and those generated with commercial HLS tools. ASSURE obfuscates the semantic information in an RTL design using approaches used in *program obfuscation* to protect the software IP [20], [22]. ASSURE obfuscates the RTL by adding in *opaque predicates* such that the evaluation of the opaque predicates depends on the locking key; their values are known to the designer during obfuscation, but unknown to the untrusted foundry. ASSURE obfuscates three **semantic** elements useful to replicate function of an IP:

- *constants* contain sensitive information in the computation (e.g., filter coefficients).
- *operations* determine functionality.
- *branches* define the execution flow (i.e., which operations are executed under conditions).

ASSURE parses the input HDL and creates the *abstract syntax tree* (AST) – step ①. It then analyzes the AST to select the semantic elements to lock (step ②) and obfuscates them (*AST elaboration* – step ③). The *RTL generation* phase (step ④) produces the output RTL design that has the same external interface as the original module, except for an additional input port that is connected to the place where $\mathcal{K}_r^*$ is stored. ASSURE starts from a synthesizable IP and modifies its description, it fits with existing EDA flows and same constraints as the original, including tools to verify that resulting RTL is equivalent to the original design when the correct key is used and to verify that it is not equivalent to the original when an incorrect key is used.

The key idea of ASSURE is that the functionality of D^* is much harder to understand without the parameter $\mathcal{K}_r^*$. If the attackers apply a key different from $\mathcal{K}_r^*$ to D^*, they obtain plausible but wrong circuits, indistinguishable from the correct one. These variants are *indistinguishable* from one another without a-priori knowledge of the design.

A. ASSURE Obfuscation Flow

To generate an obfuscated RTL design, we must match the requirements of the IP design with the constraints of the technology for storing the key (e.g., maximum size of the tamper-proof memory). On one hand, the number of bits needed to obfuscate the semantics of an RTL design depends on the complexity of the algorithm to protect. On the other hand, the maximum number of key bits that can be used by ASSURE (K_{max}) is a design constraint that depends on the technology for storing them in the circuit. ASSURE analyzes

Algorithm 1: ASSURE obfuscation.

```
1  Procedure ObfuscateModule(AST_m, K_r^*, K_max)
       Data: AST_m is the AST of the module m to obfuscate
       Data: K_r^* is the current locking key
       Data: K_max is the maximum number of key bits to use
       Result: AST_m^* is the obfuscated AST of the module m
       Result: K_r^* is the updated locking key
2      BlackList ← CreateBlackList(AST_m);
3      AST_m^* ← BlackList;
4      ObfElem ← DepthFirstAST(AST_m) \ BlackList;
5      foreach el ∈ ObfElem do
6          b_el ← BitReq(el);
7          if KeyLength(K_r^*)+b_el > K_max then
8              AST_m^* ← AST_m^* ∪ el;
9          else
10             K_el ← GetObfuscationKey(el);
11             AST_m^* ← AST_m^* ∪ Obfuscate(el, K_el);
12             K_r^* ← K_r^* ∪ K_el;
13     return {AST_m^*, K_r^*}
```

the input design to identify which modules and which circuit elements in modules must be protected. First, ASSURE does **depth-first analysis** of the design to *uniquify* the module hierarchy and creates a list of modules to process. In this way, *ASSURE hides the semantics of the different modules so that extracting knowledge from one instance does not necessarily leak information on all modules of the same type*.

After uniquifying the design, ASSURE analyzes the AST of each module with Algorithm 1 starting from the innermost ones. Given a hardware module, ASSURE first creates a "black list" of the elements that must be excluded from obfuscation (line 2). For example, the black list contains elements inside reset and update processes or loop induction variables (see Section III-B). The designer can also annotate the code to specify that specific regions or modules must be excluded from obfuscation (e.g., I/O processes or publicly-available IPs). The black-list elements are added unchanged to the output AST (line 3). Finally, ASSURE determines the list of AST elements to obfuscate (line 4) and process them (lines 5-12). For each element, it computes the number of bits required for obfuscation (line 6) and check if there are enough remaining key bits (line 7). If not, ASSURE does not obfuscate the element (line 8). Indeed, reusing a key bit across multiple elements as in [16] reduces the security strength of our scheme because extracting the key value for one element invalidates the obfuscation of all others sharing the same key bit. If the obfuscation is possible (lines 9-12), ASSURE generates the corresponding key bits (line 10). These bits depend on the specific obfuscation technique to be applied to the element and can be randomly generated, extracted from an input key (see Fig. 2), or extracted from the element itself (see Section III-B). ASSURE uses these key bits to obfuscate the element and the result is added to the output AST (line 11). The key bits are also added to the output locking key (line 12). We repeat this procedure for all modules until the top, which will return the AST of the entire design and the final key.

B. ASSURE Obfuscations and Security Proofs

Each of the ASSURE techniques targets an essential element to protect and uses a distinct part of the r-bit locking key $\mathcal{K}_r^*$, to create an opaque predicate[1]. In software, an *opaque predicate* is a predicate for which the outcome is certainly known by the programmer, but requires an evaluation at run time [20]. We create **hardware opaque predicates**, for which the outcome is determined by ASSURE (and so known) at design time, but requires to provide the correct key at run time. Any predicate involving the extra parameter $\mathcal{K}_r^*$ meets this requirement. Given a locking key $\mathcal{K}_r^*$, *ASSURE generates a circuit **indistinguishable** from the ones generated with any other $\mathcal{K}_r \neq \mathcal{K}_r^*$ when the attacker has no prior information on the design*.

We show that ASSURE techniques offer provable security guarantees [26]. Consider an m-input n-output Boolean function $\mathcal{F} : X \rightarrow Y$, where $X \in \{0,1\}^m$ and $Y \in \{0,1\}^n$. Obfuscation $\mathcal{L}$ receives $\mathcal{F}$) and an r-bit key $\mathcal{K}_r^*$ and generates a locked design $\mathcal{C}_{lock}$.

Definition An obfuscation scheme $\mathcal{L}$ is defined as:

$$\mathcal{L}(\mathcal{F}(X), \mathcal{K}_r^*) = \mathcal{C}_{lock}(X, K) \quad (1)$$

where the mapping $\mathcal{C}_{lock}$: $X \times K \rightarrow Y$, and $K \in \{0,1\}^r$ such that

- $\mathcal{C}_{lock}(X, \mathcal{K}_r^*) = \mathcal{F}_{K_r^*}(X) = \mathcal{F}(X)$
- $\mathcal{C}_{lock}(X, \mathcal{K}_r) = \mathcal{F}_{K_r}(X) \neq \mathcal{F}(X)$ when $\mathcal{K}_r \neq \mathcal{K}_r^*$

This definition shows $\mathcal{C}_{lock}$ can generate a family of Boolean functions $\{\mathcal{F}_{K_r}\}$ based on the r-bit key value $\mathcal{K}_r$. The functionality $\mathcal{F}(X)$ can only be unlocked uniquely with the correct key $\mathcal{K}_r^*$. This is followed by a corollary about an important characteristic of the family of Boolean functions that can be generated by $\mathcal{C}_{lock}(X, K)$.

Theorem 1. *For an obfuscated netlist $C_{lock}(X, K)$ created using $\mathcal{K}_r^*$ and $\mathcal{F}(X)$, the unlocked functionalities $\mathcal{F}_{K_1}$ and $\mathcal{F}_{K_2}$ (corresponding to keys K_1 and K_2) relate as follows:*

$$\mathcal{F}_{K_1} \neq \mathcal{F}_{K_2} \forall K_1, K_2 \in K, K_1 \neq K_2 \quad (2)$$

Proof. Let us first consider case (i) $K_1 = \mathcal{K}_r^*$. Therefore, by the definition of RTL obfuscation scheme $\mathcal{L}$, $\mathcal{F}_{K_1} \neq \mathcal{F}_{K_2} \forall K_2 \in K, K_1 \neq K_2$. Now, for case (ii) $K_1 \neq \mathcal{K}_r^*$, there exists a locked netlist $\mathcal{C'}_{lock}$, which locked $\mathcal{F}_{K_1}$ using K_1. Therefore, $\mathcal{F}_{K_2} = \mathcal{C'}_{lock}(X, K_2)$. By the definition of logic locking security, $\mathcal{F}_{K_2} \neq \mathcal{F}_{K_1} \forall K_2 \neq K_1$ in $\mathcal{C'}_{lock}(X, Y)$. ∎

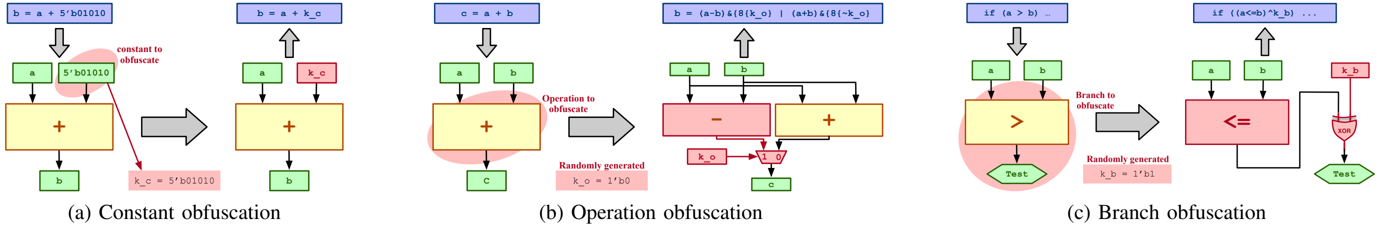
We define $P[\mathcal{C}_{lock}(X, K) | \mathcal{L}(\mathcal{F}(X), \mathcal{K}_r^*)]$ as the probability of obtaining the locked design $\mathcal{C}_{lock}(X, K)$ given that we locked the Boolean function $\mathcal{F}(X)$ applying $\mathcal{L}$ with $\mathcal{K}_r^*$. We propose the logic locking scheme $\mathcal{L}$ is secure under the oracle-less threat model as follows:

Theorem 2. *A logic locking scheme $\mathcal{L}$ for r-bit key K is secure for a family of Boolean functions $\mathcal{F}_{K_r}$ of cardinality 2^r if the following condition holds true:*

$$P[\mathcal{C}_{lock}(X, K) | \mathcal{L}(\mathcal{F}(X), \mathcal{K}_r^*)] = P[\mathcal{C}_{lock}(X, K) | \mathcal{L}(\mathcal{F}_{K_r}(X), \mathcal{K}_r)] \forall \mathcal{K}_r \neq \mathcal{K}_r^*, \mathcal{F}(X) \neq \mathcal{F}_{K_r}(X) \quad (3)$$

This theorem states that the locked netlist generated by applying logic locking scheme $\mathcal{L}$ is equally likely to be created by

[1] We use Verilog notation in the examples, but the approach is general.



(a) Constant obfuscation (b) Operation obfuscation (c) Branch obfuscation

Fig. 3: Three ASSURE obfuscations. (a) Constant (b) Branch and (c) Operation. Each obfuscation uses a portion of the key.

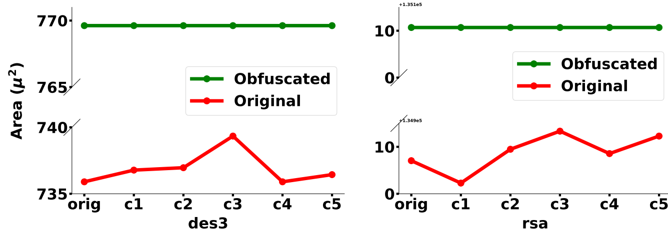


Fig. 4: Area overhead for original and obfuscated variants of benchmarks `DES3` and `RSA` having different constants ($c1-c5$).

a family of Boolean function $\mathcal{F}_{K_r}(X)$ along with the original Boolean function $\mathcal{F}(X)$. We show that above two claims are satisfied for all our obfuscation schemes and provide us a security guarantee of 2^r under the proposed threat model.

1) Constant Obfuscation: This obfuscation removes selected constants and moves them into the locking key K as shown in Fig. 3a. The original function is preserved only when the key provides the correct constant values. *Each constant bit is a* hardware-opaque predicate*; the designer knows its value and the circuit operation depends on it.*

Example: *Consider the RTL operation* `b = a + 5'b01010`. *To obfuscate the constant, we add a 5 bit key* `K_c = 5'b01010`. *The RTL is rewritten as* `b = a + K_c`. *The attacker has no extra information and* 2^5 *possibilities from which to guess the correct value.* □

Hiding constant values allows designers to protect proprietary information but also may prevent subsequent logic optimizations (e.g., constant propagation and wire trimming). However, several constants are unuseful and, in some cases, problematic to protect. For example, reset values are set at the beginning of the computation to a value that is usually zero and then assigned with algorithm-related values. Also, obfuscating reset polarity or clock sensitivity edges of the processes introduces two problems: incorrect register inferencing, which leads to synthesis issues of the obfuscated designs, and incorrect reset process that easily leads to identify the correct key value. In particular, if we apply obfuscation to the reset processes and the attacker provides an incorrect key value, the IC will be stalling in the reset state when it is supposed to be in normal execution. So, we exclude constants related to reset processes and sensitivity values for obfuscation.

Proof. The structure of the obfuscated circuit is independent of the constant and, given an r-bit constant, the 2^r values are indistinguishable. The attacker cannot get insights on the constants from the circuit structure. ASSURE constant obfuscation satisfies the provable security criteria of logic locking $\mathcal{L}$ under strong adversarial model as defined in Theorem 2.

Let us consider an RTL design of m inputs and n outputs $R: X \rightarrow Y$, $X \in \{0,1\}^m$ and uses an r-bit constant C_{orig}. ASSURE constant obfuscation converts the r-bit constant into an r-bit key K_r^* as a lock $\mathcal{L}$ and uses it to lock the design $C_{lock}(X,K)$. The obfuscated RTL is depicted as follows:

$$C_{output} = K \tag{4}$$

where, $C_{output} = C_{orig}$, when $K = K_r^* = C_{orig}$.

Claim 1: Any unlocked constant C_{K_1} and C_{K_2} using r-bit keys K_1 and K_2 are unique. (Theorem 1)

Proof. $\forall K_1 \neq K_2,\ K_1, K_2 \in \{0,1\}^r$
$\implies C_{K_1} \neq C_{K_2}$. ■

Claim 2: A constant-obfuscated circuit with r-bit key K can be generated from 2^r possible constants (each of r-bit) with equal probability, i.e. the following holds true.

$$P[C_{output}|K = K_r^*] = P[C_{output}|K = K_r] \\ \forall K_r \neq K_r^*; K_r \in 2^r \tag{5}$$

Proof. The probability of choosing K_r is uniform. So, P[$K = K_r^*$] = P[$K = K_r$], $\forall K_r \neq K_r^*$
$\implies P[C_{orig}] = P[C_r],\ C_{orig} \neq C_r, \forall C_r \in \{0,1\}^r$. ■

Claims 1 and 2 jointly denote that the constant obfuscated by 2^r unique constants are indistinguishable and can be unlocked uniquely by the correct r-bit key. Constant obfuscation hides the original constants with a security strength of 2^r.

In Fig. 4, we show area overhead of `DES3` and `RSA`, two CEP benchmarks [30]. This experiment shows that constant obfuscation generates indistinguishable circuits. We consider a variable from each benchmark: *sel_round* from `DES3` and *modulus_m1_len* from `RSA`. We generate different circuits by assigning different constants to the same variable. We synthesize these circuit variants and obtain the area overhead. Fig. 4 shows that every constant value ($c1-c5$) can be reverse engineered from the synthesized circuit since each constant directly maps to unique area overhead. On the contrary, the area overhead of synthesized circuits remain the same after obfuscation, and the obfuscated circuits are indistinguishable, making it difficult for the attacker to recover the constant.

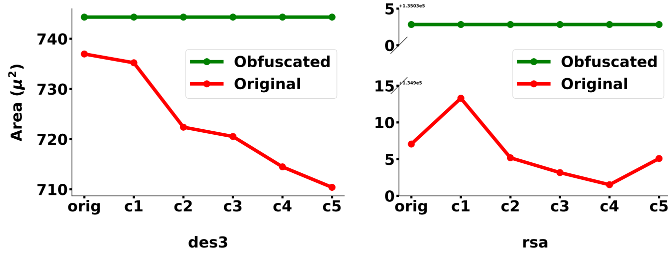


Fig. 5: Area overhead for original and obfuscated variants of DES3 and RSA using different operators in the statement.

2) Operation Obfuscation: We generate a random key bit and use it to multiplex the operation result with that from another operation sharing the same inputs, as shown in Fig. 3b. *The mux selector is a* hardware opaque predicate *because the designer knows its value and the mux propagates the correct result only for the correct key bit*. This is similar to that proposed for C- and HLS obfuscation [16], [31].

Example: *Let us obfuscate RTL operation* `c = a + b` *with a dummy subtraction. We generate a key bit* `k_0 = 1'b0` *and rewrite the RTL as* `c = k_o ? a - b : a + b`. *The original function is selected for the correct* `k_o`. □

The ternary operator is a simple representation of the multiplexer, but it may impact code coverage. It introduces extra branches in the circuit, where one of the paths is never activated once the key is provided. To keep the same coverage as the original design, we rewrite the mux selection as `o = in1 & k | in2 & ~k`.

Example: *Operation* `c = a + b` *obfuscated as* `c = k_o ? a - b : a + b` *can be written as* `c = (a - b)&{8{k_o}} | (a + b)&{8{~k_o}}`. *This is equivalent to ternary operation without branches, and same code coverage.* □

Since operations use the same inputs, ASSURE adds a multiplexer at the output with its select connected to the key bits. The multiplexer and the additional operator are area overhead. The multiplexer impacts the critical path and the additional operation introduces a delay when it takes more time than the original one. We create a pool of alternatives for each operation type. Original and dummy operations are "balanced" in complexity to avoid increasing the area and the critical path. Dummy operations are selected to avoid structures the attacker can easily unlock. Incrementing a signal by one cannot be obfuscated by a multiplication by one, clearly a fake. Dummy operators are also selected to avoid collisions. For example, adding a constant to a signal cannot be obfuscated with a subtract because the wrong operation key bit can activate the circuit when the attacker provides the two's complement of the constant.

Proof. Consider an RTL design with m inputs and n outputs, with a mapping $\mathcal{F} : X \rightarrow Y$, $X \in \{0,1\}^m$ and with r possible sites for operator obfuscation. ASSURE obfuscation uses multiplexer (MUX) based locking $\mathcal{L}$ and uses an r-bit key K_r^* to lock the design $C_{lock}(X, K)$.

$$\begin{aligned}\mathcal{C}_{lock}(X, K) &= \mathcal{F}(X, k_1, k_2, .., k_r)\\ &= \overline{k_1}\mathcal{F}(X, 0, k_2, .., k_r) + k_1\mathcal{F}(X, 1, k_2, .., k_r)\\ &= \mathcal{K}_r^1 \underbrace{\mathcal{F}(X, K = \mathcal{K}_r^1)}_{\mathcal{F}_{K_1}} + \mathcal{K}_r^2 \underbrace{\mathcal{F}(X, K = \mathcal{K}_r^2)}_{\mathcal{F}_{K_2}} + ..\\ &\quad .. + \mathcal{K}_r^{2^r} \underbrace{\mathcal{F}(X, K = \mathcal{K}_r^{2^r})}_{\mathcal{F}_{K_{2^r}}}\end{aligned} \tag{6}$$

where, $\mathcal{F}_{K_r^*}(X) = \mathcal{C}_{lock}(X, K = \mathcal{K}_r^*)$, $\mathcal{K}_r^*$ is r-bit key. Each location of operator obfuscation applies output of different operations (one original and another fake) to a multiplexer. The following equation holds true for operator obfuscation.

$$\mathcal{F}(X, k_1, .., k_i = 0, .., k_r) \neq \mathcal{F}(X, k_1, .., k_i = 1, .., k_r) \quad \forall i \in [1, r] \tag{7}$$

Secondly, the sites of operation obfuscation are different. The output of multiplexer using any key-bit value at one location is independent of the choice made elsewhere. Given a key K, the unlocked function of two circuits will be different if we set same logic value at two different key-bit locations. For an example $K = 1101$, if one chooses bit location 2 and 4 and flip them, i.e. $K_1 = 1001, K_2 = 1100$, then $F_{K_1} \neq F_{K_2}$.

$$\mathcal{F}(X, k_1, .., k_i = \overline{k_i}, .., k_r) \neq \mathcal{F}(X, k_1, .., k_j = \overline{k_j}, .., k_r) \quad \forall i, j \in [1, r], i \neq j \tag{8}$$

Claim 1: Any pair of unlocked circuit $F_{K_r^1}$ and $F_{K_r^2}$ using r-bit keys K_r^1 and K_r^2 on MUX based obfuscated circuit $C_{lock}(X, K)$ are unique. (Theorem 1)

Proof. $\forall K_r^1 \neq K_r^2$, $K_r^1, K_r^2 \in \{0,1\}^r$
$\implies$ Hamming distance $(K_1, K_2) \in [1, r]$.
$\implies$ Eq. 7 + Eq. 8, $F_{K_1} \neq F_{K_2}$ ■

Claim 2: MUX-based obfuscation with r-bit key K can be generated from r different locations having 2^r operations with equal probability, i.e. following condition holds true.

$$P[C_{lock}|(F_{K_r^*}, K_r^*)] = P[C_{lock}|F_{K_r^i}, K_r^i)] \quad \forall K_r^i \neq K_r^*; F_{K_r^i} \neq F_{K_r^*}; i \in [1, 2^r]$$

Proof. The probability of choosing K_r is uniform. Therefore, P[K = K_r^*] = P[K = K_r^i], $\forall K_r^i \neq K_r^*$
$\implies P[C_{lock}(X, K = K_r^*]) = P[C_{lock}(X, K = K_r^i)]$
$\implies P[F_{K_r^*}] = P[F_{K_r^i}] = \frac{1}{2^r}$. ■

Claims 1 and 2 show that operator obfuscation can generate indistinguishable netlists.

In Fig. 5, we demonstrate area overhead of the two benchmark circuits *DES3* and *RSA* for operator obfuscation supporting our claims generate indistinguishable circuits. We consider a single operation from each benchmark: addition of *auxiliary input* and *round_output* from *DES3*, and subtraction of *modulus_m1_len* from a constant value in *RSA*. We generate different circuits by replacing the original operators with other operators. After synthesis, area overhead of these variants (Fig. 5) are unique and can be reverse engineered. On the contrary, the area overhead of synthesized circuits remain the same after obfuscation and so the obfuscated circuits reveals nothing about the original operator.

3) Branch Obfuscation: To hide which branch is taken after the evaluation of an RTL condition, we obfuscate the test with a key bit as `cond_res⊕k_b`, as shown in Fig. 3c. To maintain semantic equivalence, we negate the condition to reproduce the correct control flow when `k_b = 1'b1` because the XOR gate inverts the value of `cond_res`. We apply De Morgan's law to propagate the negation to disguise the identification of the correct condition. *The resulting predicate is* hardware-opaque *because the designer knows which branch is taken but this is unknown without the correct key bit.*

Example: *Let* `a > b` *be the RTL condition to obfuscate with key* `k_b = 1'b1`*. We rewrite the condition as* `(a <= b)∧k_b`*, which is equivalent to the original one only for the correct key bit. The attacker has no additional information to infer if the original condition is > or <=.* □

Obfuscating a branch introduces a 1-bit XOR gate, so the area and delay effects are minimal. Similar to constant obfuscation, branch obfuscation is applied only when relevant. For example, we do not obfuscate reset and update processes. We apply the same technique to ternary operators. When these operators are RTL multiplexers, this technique thwarts the data propagation between the inputs and the output. The multiplexer propagates the correct value with the correct key.

Proof. For an m input RTL design, we have a control-flow graph (CFG) $G(C,E)$ having $|C|$ nodes and $|E|$ edges. We do a depth-first-traversal of the CFG and order the r conditional nodes in the way they are visited. Let the ordered set of conditional nodes be $C_{orig} = \{c_1, c_2, ...c_r\}$ ($r = |C|$). ASSURE branch obfuscation xor C_{orig} with r-bit key K_r^* as the logic locking scheme $\mathcal{L}$ and generate a locked design $G(C_{encrypted}, K)$. For eg. if $C_{orig} = \{c_1, c_2, c_3, c_4\}$ and $K = 1101$, then $C_{encrypted} = \{\overline{c_1}, \overline{c_2}, c_3, \overline{c_4}\}$. The locked design, post branch-obfuscation is illustrated as follows.

$$G(C_{encrypted}, E, K) = G(C_{orig} \oplus K_r^*, E) \quad (9)$$

where $G(C_{orig}, E) = G(C_{encrypted}, K = K_r^*, E) = G(C_{encrypted} \oplus K_r^*, E)$.

Claim 1: Any unlocked CFG $G(C_{K_1}, E)$ and $G(C_{K_2}, E)$ using r-bit keys K_1 and K_2 on XOR based encrypted CFG $G(C_{encrypted}, K, E)$ are unique. (Theorem 1)

Proof. $\forall K_1 \neq K_2,\ K_1, K_2 \in \{0,1\}^r$
$\implies K_1 \oplus C_{encrypted} \neq K_2 \oplus C_{encrypted} \implies C_{K_1} \neq C_{K_2}$.
$\implies G(C_{K_1}, E) \neq G(C_{K_2}, E)$. ■

Claim 2: CFG obfuscated design $G(C_{encrypted}, E, K)$ can be generated from 2^r possible combination of condition statuses with equal probability, i.e. the following condition holds true.

$$P[G(C_{encrypted}, E, K) | G(C_{orig} \oplus K_r^*, E)] = P[G(C_{encrypted}, E, K) | G(C_r \oplus K_r, E)] \quad \forall K_r \neq K_r^*; C_{orig} \neq C_r \quad (10)$$

Proof. The probability of choosing K_r is uniform. So,
$P[K = K_r^*] = P[K = K_r],\ \forall K_r \neq K_r^*, K_r \in 2^r$
$\implies P[C_{encrypted} \oplus K_r^*] = P[C_{encrypted} \oplus K_r]$
$\implies P[C_{orig}] = P[C_r]\ C_{orig} \neq C_r$,
$C_r = \{p_1, p_2, .., p_i, .., p_r\}$, where $p_i \in \{c_i, \overline{c_i}\}$. ■

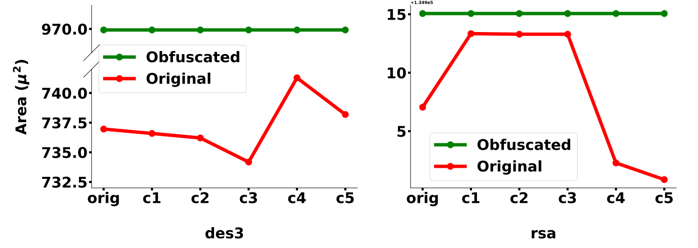


Fig. 6: Area overhead for original and obfuscated variants of benchmarks `DES3` and `RSA` having different CFG flows.

Combining claims 1 and 2 shows that the encrypted CFGs are indistinguishable for a family of 2^r possible designs.

In Fig. 6, we report the area overhead of the two benchmark circuits `DES3` and `RSA` in case of branch obfuscation showing empirical evidence of our claim that obfuscated circuits are indistinguishable. We identify five conditions from each benchmark and generated five different variants, flipping each condition at a time. After synthesizing the circuits, we observed that area overhead is uniquely mapped to each variant of the design. The conditions in the CFG can be easily reverse engineered from the synthesized circuit and the flow of design can be unlocked. On the contrary, the area overhead of synthesized circuits remain the same after obfuscation, indicating the obfuscated circuits reveal no information about the control-flow of the circuit.

C. ASSURE Against Oracle-Less Attacks

We outlined provable security guarantees of ASSURE's RTL obfuscation technique via design indistinguishability. In earlier section, we show that for n-bit obfuscation technique there are 2^n possible RTL designs which can generate same obfuscated circuit. Using the proofs we have provided for ASSURE's obfuscation scheme, we show the resilience of ASSURE against state-of-art oracle-less attacks.

1) Resilience against desynthesis and redundancy attacks: Massad *et al.* [26] showed that greedy heuristics can recover the key of an obfuscated circuit post logic synthesis. An incorrect key assignment results in large redundancy in the circuit triggering additional optimizations when re-synthesized. Similarly, Li *et al.* [27] propose an oracle-less attack using concepts from VLSI testing. Incorrect key results in large logic redundancy and most of stuck-at faults become untestable. A correctly unlocked circuit however has high testability. ASSURE obfuscates the design at the RTL followed by synthesis. Since, the obfuscated RTL is equally likely to be generated from 2^n possible designs (for n-bit obfuscation), logic synthesis using different keys on a reverse-engineered obfuscated netlist reveals no information about the original netlist. Hence, the area overhead for the correct and incorrect keys are in same range (see Figs. 4, 5 and 6).

2) Resilience against ML-guided attacks: Chakraborty *et al.* [28], [29] proposed oracle-less attacks on logic obfuscation by exploiting the fact that obfuscation techniques hide the functional by inserting XOR/XNOR gates and the process leaves traces of the structural signature. The key gates are

TABLE I: Characteristics of the input RTL benchmarks.

Suite	Design	Modules	Const	Ops	Branches	Tot Bits	Comb cells	Seq cells	Buf cells	Inv cells	# nets	Area (μm^2)	Delay (ns)
CEP	AES	657	102,403	429	1	819,726	127667	8502	506	21812	136493	42854.69	136.75
	DES3	11	4	3	775	898	2076	135	128	368	2448	736.96	192.28
	DFT	211	447	151	132	8,697	118201	38521	9552	41320	158807	81865.94	336.72
	FIR	5	10	24	0	344	820	439	49	225	1704	1129.36	377.76
	IDFT	211	447	151	132	8,697	118154	38525	9576	41305	158722	81821.90	333.59
	IIR	5	19	43	0	651	1378	648	72	367	2621	1679.72	464.82
	MD5	2	150	50	1	4,533	4682	269	168	923	5756	1840.15	791.53
	RSA	15	243	35	13	1,942	222026	57987	21808	66088	280222	134907.05	386.55
	SHA256	3	159	36	2	4,992	5574	1040	243	1024	7532	3201.07	440.67
IWLS	MEM_CTRL	27	492	442	160	2,096	4007	1051	120	1136	5183	2373.35	260.72
	SASC	3	35	27	17	126	367	116	0	125	500	238.24	84.4
	SIMPLE_SPI	3	55	34	15	288	476	130	2	145	623	282.57	119.42
	SS_PCM	1	5	10	3	24	231	87	1	94	338	168.29	90.51
	USB_PHY	3	67	70	34	223	287	98	0	85	401	194.15	71.91
OpenCores	ETHMAC	66	487	1,217	218	3,849	34783	10545	2195	12021	45441	22453.76	190.44
	I2C_SLAVE	4	104	14	11	269	466	125	0	126	596	160.28	125.44
	VGA_LCD	16	123	310	56	885	54614	17052	4921	19228	71766	36095.90	224.67
OpenROAD	ARIANE_ID	4	3,498	385	723	4,606	1993	378	96	559	2615	980.97	225.48
	GCD	11	15	4	12	496	168	34	3	32	253	100.91	161.87
	IBEX	15	14,740	5,815	6,330	26,885	12161	1864	978	2965	14379	5758.84	538.1

assumed inserted into the design before synthesis, and the technology library and synthesis algorithm/tool are known. Since the effect of logic optimizations remains local and optimization rules are deterministic, ML models can reconstruct the pre-synthesis design from an obfuscated circuit. One can recover the original function by launching an ML-guided removal attack on obfuscated RTL. In ASSURE, the obfuscation logic does not depend solely on insertion of XOR/XNORs. For example, in branch obfuscation, we do logic inversion instead of simple XOR followed by NOT when keybit=1. Recovering the original RTL from obfuscated RTL is hard (see claim 2 of ASSURE branch obfuscation proof in Section III-B3).

IV. Experimental Validation of ASSURE

A. Experimental Setup

We implemented ASSURE as a Verilog→Verilog tool that leverages Pyverilog [32], a Python-based hardware design processing toolkit to manipulate RTL Verilog. Pyverilog parses the input Verilog descriptions and creates the design AST. ASSURE then manipulates the AST. Pyverilog is then used to re the output Verilog description ready for logic synthesis.

We used ASSURE to protect several Verilog designs from different sources[2]: the MIT-LL Common Evaluation Platform (CEP) platform [30], the OpenROAD project [33], and the OpenCores repository [34]. Four CEP benchmarks (DCT, IDCT, FIR, IIR) are created with Spiral, a hardware generator [35]). Table I shows the characteristics of these benchmarks in terms of number of hardware modules, constants, operations, and branches. This data also characterizes the functionality that needs obfuscation. The benchmarks are much larger than those used by the gate-level logic locking experiments by the community [9]. Differently from [16], ASSURE does not require any modifications to tools and applies to pre-existing industrial designs without access to an HLS tool. ASSURE processes the Verilog RTL descriptions with no modifications.

[2]Supporting VHDL and SystemVerilog only requires proper HDL parsers.

We analyzed the ASSURE in terms of security (Section IV-B) and overhead (Section IV-C). For each benchmark, we created obfuscated variants using all techniques (`ALL`) or one of constant (`CONST`), operation (`OP`), and branch (`BRANCH`) obfuscations. We repeat experiments by constraining the number of key bits available: 25%, 50%, 75% or 100% and reported in Table I. The resulting design is then identified by a combination of its name, the configuration, and the number of key bits. For example, `DFT-ALL-25` indicates obfuscation of the `DFT` benchmark, where all three obfuscations are applied using 2,175 bits for obfuscation (25% of 8,697) as follows: 38 for operations (25% of 151), 33 for branches (25% of 132) and the rest (2,104) for constants.

B. Security Assessment

Since no activated IC is available to the attacker (see Section II), we can only use methods based on the application of random keys to analyze the security of our techniques for thwarting reverse engineering of the IC functionality [10]. We base our experimental analysis on formal verification of the locked design against the unprotected design. The goal is twofold. First, we show that, when the correct key $\mathcal{K}_r^*$ is used, the unlocked circuit matches the original. We label this experiment as CORRECTNESS. Second, we show that flipping each single key bit induces at least a failing point (i.e., no *collision*). This experiment demonstrates that each key bit has an effect on the functionality of the circuit. We label this experiment as KEY EFFECT. We show that there is no other key that can activate the same IC. In this experiment, we also aim at quantifying how the obfuscation techniques affect the IC functionality when the attacker provides incorrect keys. We compute the *verification failure* metric defined as follows:

$$F = \frac{1}{K} \cdot \sum_{i=1}^{K} \frac{n(FailingPoints)_i}{n(TotalPoints)} \quad (11)$$

This metric is the average fraction of verification points that do not match when testing with different wrong keys. We experimented using Synopsys Formality N-2017.09-SP3.

Fig. 7: Verification failure metric in KEY-EFFECT experiments.

1) Correctness: We apply ASSURE several times, each time with a random key to obfuscate operations and branches[3]. We formally verified these designs against the original ones. In all experiments, *ASSURE generates circuits that match the original design with the correct key*.

2) Key Effect: Given a design obfuscated with an r-bit key, we performed r experiments where in each of them with flipped one and only one key bit with respect to the correct key. In all cases, *formal verification identifies at least one failing point, showing that an incorrect key always alters the circuit functionality*. Also in this case, varying the locking key has no effect since the failure is induced by the flipped bit (from correct to incorrect) and not its value. Fig. 7 shows the verification failure metrics for each experiment. Results are not reported for `FIR-BRANCH-*` and `IIR-BRANCH-*` because they have no branches. `AES`, `DFT`, `IDFT`, and `OPENCORES-ETHMAC` benchmarks have low values ($\sim 10^{-5}$) because these benchmarks have many verification points and only a small part is failing. Operations and constants vitally impact the design as obfuscating them induces more failing points. Increasing the number of obfuscated operations reduces the metric. Since obfuscation is performed using a depth-first analysis, the first bits correspond to operations closer to the inputs. As the analysis proceeds the obfuscation is closer to the output and more internal points match. The metric is an average across all cases. When all elements are obfuscated, these effects are averaged.

This experiment allowed us to identify design practices that lead to inefficient obfuscations or even collisions. In `DFT`, one-bit signals were initialized with 32-bit integers with values 0/1. While Verilog allows this syntax, the signals are trimmed by logic synthesis. A naive RTL constant analysis would pick 32 bits for obfuscating a single-bit. Since only the least significant bit impacts the circuit function, flipping the other 31 bits would lead to a collision. So, we extended ASSURE AST analysis to match the constant sizes with those of the target signals.

C. Synthesis Overhead

We did logic synthesis using the Synopsys Design Compiler J-2018.04-SP5 targeting the Nangate 15nm ASIC technology at standard operating conditions (25C). We evaluated the area overhead and critical-path delay degradation relative to the original design. While our goal is to protect the IP functionality and not to optimize the resources, designs with lower cost are preferred. *ASSURE generates correct designs with no combinational loops*. Constant obfuscation extracts the values that are used as the key and no extra logic. Operation obfuscation multiplexes results of original and dummy operations. Branch obfuscation adds XOR to the conditions.

[3]Constants are always extracted in the same way.

Fig. 8: Area overhead for ASSURE obfuscation. Benchmarks are presented in increasing order of total overhead.

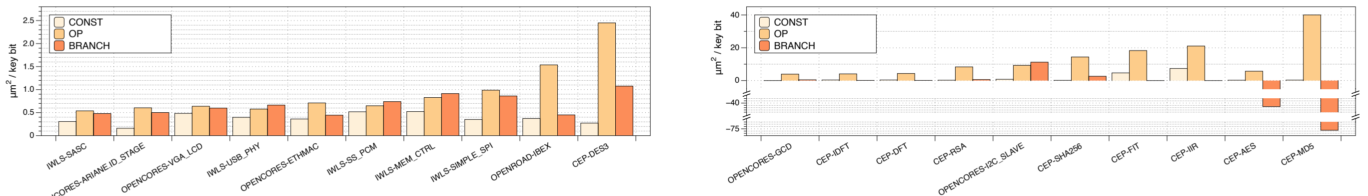


Fig. 9: Area overhead per key bit for ASSURE obfuscation. Benchmarks are presented in increasing order of total overhead.

1) Area overhead: Table I reports the results of the original design – the number of cells in the netlists, the area (in μm^2) and the critical-path delay (in ns). Fig. 9 reports the area overhead of all obfuscations with respect to the original designs. The three techniques are independent and so, `ALL` results are the aggregate of the three techniques. Constant obfuscation produces an average overhead in the range 18% (`⋆-CONST-25`) to 80% (`⋆-CONST-100`). The maximum overhead is about 450% for `AES-CONST-100`, which has the most number obfuscated constants. *ASSURE removes hard-coded constants from the circuit, preventing logic optimizations like constant propagation*. The average operation obfuscation overhead is in the range 9% (`⋆-OP-25`) to 25% (`⋆-OP-100`). `IBEX-OP-100` has the maximum overhead of 155% since it has the most operations. Branch obfuscation produces a smaller average overhead, in the range 6% (`⋆-BRANCH-25`) to 14% (`⋆-BRANCH-100`) with a maximum overhead of 113% for `DES-BRANCH-100`. This benchmark has the largest proportion of branches relative to other elements. `MD5` results in savings (∼4%) when we apply branch obfuscation (`MD5-BRANCH-⋆`). The branch conditions help pick elements from the library that lower area overhead.

The real impact of ASSURE depends on how many elements are obfuscated in each configuration. So, we computed the *area overhead per key bit* as the area overhead of a configuration divided by the number of key bits used for its obfuscation and report it in Fig. 8. In most cases, *operation obfuscation has the largest impact, followed by branches and then constants*. This impact is larger for data-intensive benchmarks, like CEP filters (`DFT`, `IDFT`, `FIR`, and `IIR`). Constants usually require more obfuscation bits, so the impact per bit is smaller. Each obfuscated operation introduces a new functional unit and multiplexer per key bit. `MD5` has a large negative impact when obfuscating the branches justifying

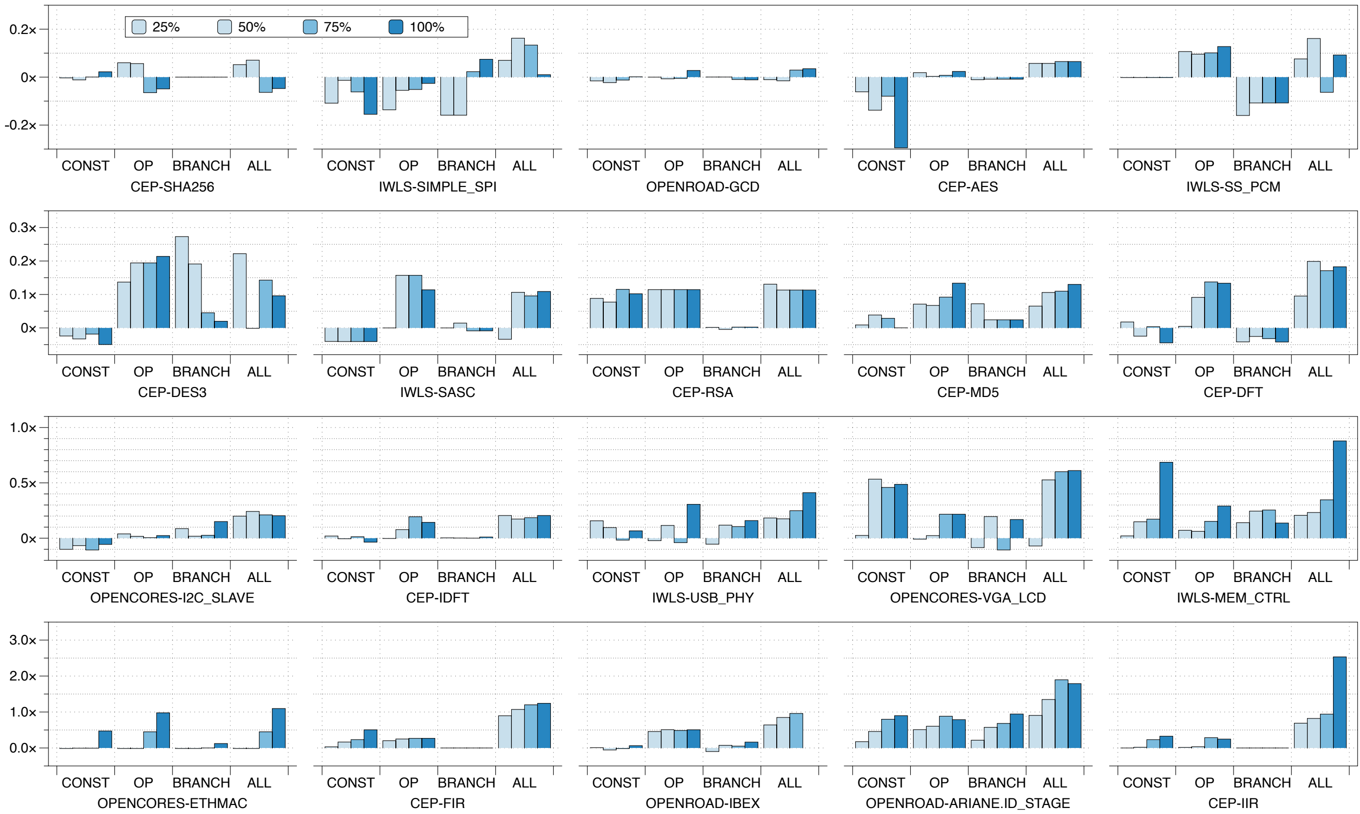


Fig. 10: Timing overhead for ASSURE obfuscation. Benchmarks are presented in increasing order of total overhead.

the area reduction when we apply only branch obfuscation (`MD5-BRANCH-*`). On the contrary, even if `AES` was the benchmark with the largest overhead (and many more bits), its overhead per key bit is comparable with the others. We repeated the experiments several times and we observed minimal variants with different locking keys.

To conclude *the area overhead is related to the design characteristics and to the number of key bits*. The former determine the impact of ASSURE, while the latter determine the total amount of overhead. *The overhead depends on the design, the techniques, and the number of key bits and not on the values of the locking key*.

2) Timing overhead: Fig. 10 shows the overhead introduced by the ASSURE obfuscation logic on the critical path when synthesis is performed targeting area optimization. *Timing overhead is application dependent with similar results across the different techniques. The overhead is larger when the obfuscated elements are on the critical path*. This is relevant in data-intensive (with many operations) and control-intensive (with control branches on critical path) designs. In most benchmarks, the timing overhead is $<20\%$. Constants have a positive impact on the overhead (see `AES` and `DES3`).

V. Discussion and Concluding Remarks

We presented ASSURE, an RTL locking framework against an *untrusted foundry that has no access to an unlocked functional chip*. This oracle-less threat model is relevant for low-volume IC production. ASSURE operates on the Verilog RTL description and is compatible with industrial EDA flows.

ASSURE hides the essential semantics (constants, operations, and control-flow branches) in a way that is *indistinguishible and provably secure against attackers with no prior knowledge of the IP function*, preventing oracle-less attacks. In our experimental analysis with formal verification and logic synthesis EDA tools, we show the *circuits can be unlocked only with the correct key* and *obfuscating the design closer to the inputs induces more verification failures*. ASSURE obfuscations introduce *area overhead that depends on the obfuscation techniques and is proportional to the number of key bits*. In case of constants, obfuscation prevent logic optimizations, like constant propagation, while *operation obfuscation has the largest overhead per key bit* and the *key values have no impact on the obfuscation results*. ASSURE has no impact on the clock cycles but only on the critical path delay in a way that depends on where the obfuscation is applied. These guidelines can be used by the designer to understand how to apply obfuscation on a given design.

References

[1] S. W. Jones, "Technology and Cost Trends at Advanced Nodes," IC Knowledge LLC, 2019.

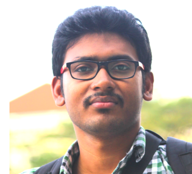
[2] J. Hurtarte, E. Wolsheimer, and L. Tafoya, *Understanding Fabless IC Technology*. Elsevier, Aug. 2007.

[3] S. Heck, S. Kaza, and D. Pinner, "Creating value in the semiconductor industry," *McKinsey on Semiconductors*, pp. 5–144, Oct. 2011.

[4] U. Guin, K. Huang, D. DiMase, J. M. Carulli, M. Tehranipoor, and Y. Makris, "Counterfeit Integrated Circuits: A rising threat in the global semiconductor supply chain," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1207–1228, Aug. 2014.

[5] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L. Wang, “Challenges and trends in modern SoC design verification,” *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, Oct. 2017.
[6] J. Rajendran, O. Sinanoglu, and R. Karri, “Is split manufacturing secure?” in *Design, Automation & Test Conference in Europe*, 2013, pp. 1259–1264.
[7] A. T. Abdel-Hamid, S. Tahar, and E. M. Aboulhamid, “IP watermarking techniques: Survey and comparison,” in *IEEE International Workshop on System-on-Chip for Real-Time Applications*, 2003, pp. 60–65.
[8] K. Shamsi, M. Li, K. Plaks, S. Fazzari, D. Z. Pan, and Y. Jin, “IP protection and supply chain security through logic obfuscation: A systematic overview,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 24, no. 6, Sep. 2019.
[9] B. Tan, R. Karri, N. Limaye, A. Sengupta, O. Sinanoglu, M. M. Rahman, S. Bhunia, D. Duvalsaint, R. Blanton, A. Rezaei, Y. Shen, H. Zhou, L. Li, A. Orailoglu, Z. Han, A. Benedetti, L. Brignone, M. Yasin, J. Rajendran, M. Zuzak, A. Srivastava, U. Guin, C. Karfa, K. Basu, V. V. Menon, M. French, P. Song, F. Stellari, G.-J. Nam, P. Gadfort, A. Althoff, J. Tostenrude, S. Fazzari, E. Breckenfeld, and K. Plaks, “Benchmarking at the frontier of hardware security: Lessons from logic locking,” *arXiv*, 2020.
[10] S. Amir, B. Shakya, X. Xu, Y. Jin, S. Bhunia, M. Tehranipoor, and D. Forte, “Development and evaluation of hardware obfuscation benchmarks,” *Journal of Hardware and Systems Security*, vol. 2, pp. 142–161, 2018.
[11] Y. Shen, Y. Li, A. Rezaei, S. Kong, D. Dlott, and H. Zhou, “BeSAT: Behavioral SAT-based Attack on Cyclic Logic Encryption,” in *Asia and South Pacific Design Automation Conference*, 2019, pp. 657–662.
[12] 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, Feb. 2019.
[13] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. J. Rajendran, and O. Sinanoglu, “Provably-secure logic locking: From theory to practice,” in *Conference on Computer and Communications Security*, 2017, pp. 1601–1618.
[14] F. Yang, M. Tang, and O. Sinanoglu, “Stripped Functionality Logic Locking with Hamming Distance Based Restore Unit (SFLL-hd) – unlocked,” *IEEE Transactions on Information Forensics and Security*, pp. 1–9, 2019.
[15] D. Sirone and P. Subramanyan, “Functional analysis attacks on logic locking,” in *Design, Automation & Test Conference in Europe*, Mar. 2019, pp. 1–6.
[16] C. Pilato, F. Regazzoni, R. Karri, and S. Garg, “TAO: Techniques for algorithm-level obfuscation during high-level synthesis,” in *Design Automation Conference*, Jun. 2018, pp. 1–6.
[17] M. Yasin, C. Zhao, and J. J. Rajendran, “SFLL-HLS: Stripped-functionality logic locking meets high-level synthesis,” in *International Conference on Computer-Aided Design*, 2019, pp. 1–4.
[18] Y. Lao and K. K. Parhi, “Obfuscating dsp circuits via high-level transformations,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 5, pp. 819–830, 2015.
[19] G. Di Crescenzo, A. Sengupta, O. Sinanoglu, and M. Yasin, “Logic locking of boolean circuits: Provable hardware-based obfuscation from a tamper-proof memory,” in *Innovative Security Solutions for Information Technology and Communications*, E. Simion and R. Géraud-Stewart, Eds. Cham: Springer International Publishing, 2020, pp. 172–192.
[20] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep. 148, 1997.
[21] C. K. Behera and D. L. Bhaskari, “Different obfuscation techniques for code protection,” in *International Conference on Eco-friendly Computing and Communication Systems*, vol. 70, 2015, pp. 757 – 763.
[22] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu, “On secure and usable program obfuscation: A survey,” *ArXiv*, 2017.
[23] K. Shamsi, D. Z. Pan, and Y. Jin, “On the impossibility of approximation-resilient circuit locking,” in *IEEE International Symposium on Hardware Oriented Security and Trust*, 2019, pp. 161–170.
[24] D. S. B. T. Force, “Report on high performance microchip supply,” https://www.hsdl.org/?abstract&did=454591, 2005.
[25] J. Rajendran, A. Ali, O. Sinanoglu, and R. Karri, “Belling the cad: Toward security-centric electronic system design,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 11, pp. 1756–1769, Nov. 2015.
[26] M. E. Massad, J. Zhang, S. Garg, and M. V. Tripunitara, “Logic locking for secure outsourced chip fabrication: A new attack and provably secure defense mechanism,” *arXiv*, 2017.
[27] L. Li and A. Orailoglu, “Piercing logic locking keys through redundancy identification,” in *Design, Automation & Test Conference in Europe*, 2019, pp. 540–545.
[28] P. Chakraborty, J. Cruz, and S. Bhunia, “SAIL: Machine learning guided structural analysis attack on hardware obfuscation,” in *Asian Hardware Oriented Security and Trust Symposium*, 2018, pp. 56–61.
[29] ——, “SURF: Joint structural functional attack on logic locking,” in *International Symposium on Hardware Oriented Security and Trust*, 2019, pp. 181–190.
[30] MIT Lincoln Laboratory, “Common Evaluation Platform (CEP),” Available at: https://github.com/mit-ll/CEP.
[31] H. Badier, J. L. Lann, P. Coussy, and G. Gogniat, “Transient key-based obfuscation for hls in an untrusted cloud environment,” in *Design, Automation & Test Conference in Europe*, 2019, pp. 1118–1123.
[32] S. Takamaeda-Yamazaki, “Pyverilog: A Python-based hardware design processing toolkit for Verilog HDL,” in *International Symposium on Applied Reconfigurable Computing*, Apr. 2015, pp. 451–460.
[33] T. Ajayi, V. A. Chhabria, M. Fogaca, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, F. Pradipta, S. Reda, M. Saligane, S. S. Sapatnekar, C. Sechen, M. Shalan, W. Swartz, L. Wang, Z. Wang, M. Woo, and B. Xu, “Toward an open-source digital flow: First learnings from the openroad project,” in *Design Automation Conference*, 2019.
[34] Oliscience, “OpenCores repository,” Available at: https://opencores.org/.
[35] M. Püschel, F. Franchetti, and Y. Voronenko, *Encyclopedia of Parallel Computing*. Springer, 2011, ch. Spiral.

Christian Pilato is a Tenure-Track Assistant Professor at Politecnico di Milano. He was a Post-doc Research Scientist at Columbia University (2013-2016) and at the ALaRI Institute of the Università della Svizzera italiana (2016-2018). He was also a Visiting Researcher at New York University, TU Delft, and Chalmers University of Technology. He has a Ph.D. in Information Technology from Politecnico di Milano (2011). His research interests include high-level synthesis, reconfigurable systems and system-on-chip architectures, with emphasis on memory and security aspects. He served as program chair of EUC 2014 and is currently serving in the program committees of many conferences on EDA, CAD, embedded systems, and reconfigurable architectures (DAC, ICCAD, DATE, CASES, FPL, ICCD, etc.) He is an IEEE Senior Member, and a member of ACM and HiPEAC.



Animesh Basak Chowdhury received his MS in Computer Science from Indian Statistical Institute in 2016. Currently, he is a doctoral candidate at the NYU Centre for Cybersecurity. His research interests include Secure Electronics Design Automation (EDA), machine learning and SoC security. Prior to joining the Ph.D. program, he spent three years as a researcher at Tata Research Development and Design Centre (TRDDC), India, where he was primarily working in the area of formal verification and security testing. He has won several awards and recognition in International Software Verification and Testing Competitions (SV-COMP, TEST COMP, and RERS-Challenge).

Donatella Sciuto received the Laurea (Ms) in Electronic Engineering from Politecnico di Milano and the PhD in Electrical and Computer Engineering from the University of Colorado, Boulder, and the MBA from Bocconi University. She is currently the Executive Vice Rector of the Politecnico di Milano and Full Professor in Computer Science and Engineering. Her main research interests cover the methodologies for the design of embedded systems and multicore systems considering performance, power and security metrics. More recently she has been involved in managing and developing research projects in the area of smart cities and in the application of new ICT technologies to different application fields. She has published over 300 scientific papers. She is a Fellow of IEEE for her contributions in embedded system design. She has served as Vice-President of Finance and then President of the IEEE Council of Electronic Design Automation from 2009 to 2013 and she serves in different capacities in IEEE Awards Committees, in scientific boards of IEEE journals and conferences.

Ramesh Karri is a Professor of ECE at New York University. He co-directs the NYU Center for Cyber Security (http://cyber.nyu.edu). He founded the Embedded Systems Challenge (https://csaw.engineering.nyu.edu/esc), the annual red team blue team event. He co-founded Trust-Hub (http://trust-hub.org). Ramesh Karri has a Ph.D. in Computer Science and Engineering, from the UC San Diego and a B.E in ECE from Andhra University. His research and education activities in hardware cybersecurity include trustworthy ICs; processors and cyber-physical systems; security-aware computer-aided design, test, verification, validation, and reliability; nano meets security; hardware security competitions, benchmarks, and metrics; biochip security; additive manufacturing security. He published over 250 articles in leading journals and conference proceedings. Karri's work on hardware cybersecurity received best paper nominations (ICCD 2015 and DFTS 2015) and awards (ACM TODAES 2018, ITC 2014, CCS 2013, DFTS 2013 and VLSI Design 2012). He received the Humboldt Fellowship and the NSF CAREER Award. He is the editor-in-chief of ACM JETC and serve(d)s on the editorial boards of IEEE and ACM Transactions (TIFS, TCAD, TODAES, ESL, D&T, JETC). He was an IEEE Computer Society Distinguished Visitor (2013-2015). He served on the Executive Committee of the IEEE/ACM DAC leading the SecurityDAC initiative (2014-2017). He served as program/general chair of conferences and serves on program committees. He is a Fellow of the IEEE for leadership and contributions to Trustworthy Hardware.

Siddharth Garg received his Ph.D. degree in Electrical and Computer Engineering from Carnegie Mellon University in 2009, and a B.Tech. degree in Electrical Engineering from the Indian Institute of Technology Madras. He joined NYU in Fall 2014 as an Assistant Professor, and prior to that, was an Assistant Professor at the University of Waterloo from 2010-2014. His general research interests are in computer engineering, and more particularly in secure, reliable and energy-efficient computing. In 2016, Siddharth was listed in Popular Science Magazine's annual list of "Brilliant 10" researchers. Siddharth has received the NSF CAREER Award (2015), and paper awards at the IEEE Symposium on Security and Privacy (S&P) 2016, USENIX Security Symposium 2013, at the Semiconductor Research Consortium TECHCON in 2010, and the International Symposium on Quality in Electronic Design (ISQED) in 2009. Siddharth also received the Angel G. Jordan Award from ECE department of Carnegie Mellon University for outstanding thesis contributions and service to the community. He serves on the technical program committee of several top conferences in the area of computer engineering and computer hardware, and has served as a reviewer for several IEEE and ACM journals.