ASSURE: RTL Locking Against an Untrusted Foundry

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

***Abstract*— Semiconductor design companies are integrating proprietary intellectual property (IP) blocks to build custom integrated circuits (ICs) 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 most of 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: 1) it allows designers to obfuscate IP cores generated with many different methods (e.g., hardware generators, high-level synthesis tools, and preexisting IPs); 2) it obfuscates the semantics of an IC before logic synthesis; and 3) it does not require modifications to EDA flows. We perform a cost and security assessment of ASSURE against state-of-the-art oracle-less attacks.**

***Index Terms*— IP protection, logic locking, register-transfer level (RTL), untrusted foundry.**

I. Introduction

THE cost of integrated circuit (IC) manufacturing has increased 5× when scaling from 90 to 7 nm [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 shows 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. Assuming a trusted design house, the foundry is the first place where a malicious attacker can reverse engineer and replicate an IC.

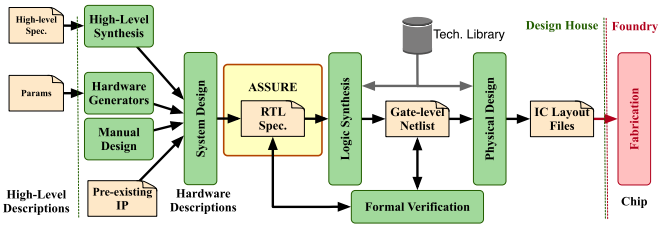


Fig. 1. State-of-the-art IC design flow. Designers create RTL description of an IC either by manual design or by using HLS tools or hardware generators. The netlist after more processing steps is sent to a third-party foundry. ASSURE locks an RTL description before logic synthesis.

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. While the design process becomes more complex, the designers cannot guarantee complete security. 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

Logic locking is a popular technique to protect the IP of ICs [9]. Designers can apply logic locking at different abstraction levels and configure the protection based on the information available to the attacker [8].

Many existing methods operate on the gate-level netlists [9]. Gate-level locking cannot obfuscate all the semantic information because logic synthesis and optimizations absorb much of

Manuscript received October 11, 2020; revised January 28, 2021 and March 21, 2021; accepted April 6, 2021. Date of publication May 10, 2021; date of current version June 29, 2021. This work was supported in part by the NSF Award under Grant 1526405, in part by the Office of Naval Research (ONR) Award under Grant N00014-18-1-2058, in part by the NSF CAREER Award under Grant 1553419, in part by the NYU Center for Cybersecurity, and in part by the NYUAD Center for Cybersecurity. *(Corresponding author: Christian Pilato.)*

Christian Pilato and Donatella Sciuto are with the Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, 20133 Milan, Italy (e-mail: christian.pilato@polimi.it; donatella.sciuto@polimi.it).

Animesh Basak Chowdhury, Siddharth Garg, and Ramesh Karri are with the NYU Center for Cybersecurity, New York University, New York, NY 11201 USA (e-mail: abc586@nyu.edu; garg@nyu.edu; rkarri@nyu.edu).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2021.3074004.

Digital Object Identifier 10.1109/TVLSI.2021.3074004

it into the netlist before the locking step. For example, constant propagation absorbs and propagates the constants. Our method completely strips the constants from the circuit before synthesis. Recently, alternative high-level locking methods obfuscate the semantic information before logic optimizations embed them into the netlist [10], [11]. For example, TAO applies obfuscations during HLS [10] but requires access to the HLS source code to integrate the obfuscations and cannot obfuscate existing IPs. Protecting a design at the RTL is an interesting compromise [12], [13]. 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. Lao and Parhi [12] proposed structural and functional obfuscation for DSP circuits. We propose a more general method that can be applied to any type of circuit. Chakraborty and Bhunia [13] proposed a method to insert a special finite-state machine to control the transition between obfuscated mode (incorrect function) and normal mode (correct function). Such transition can only happen with a specific input sequence. We use a similar method to obfuscate the operations without additional logic (and power-up overhead) to make the circuit functional in normal mode. To obfuscate the semantic information, ASSURE leverages prior work on software program obfuscation [14]–[16]. These methods obfuscate data structures, control flows, and constants through code transformations or by loading information from memory at runtime. We use a similar approach to create opaque predicates dependent only on the locking key [13].

When the attackers have access only to the circuit netlist (like in the early stages of the fabrication process), they need to identify the correct variant among the ones created by the locking key. Redundancy attacks can recover part of the key bits for several locking methods [17], while machine learning can predict the key bit values based on the structure of the circuit [18]. However, such attacks cannot recover what is not present in the circuit, like extracted constants, and cannot distinguish semantically equivalent variants. When the attackers can access also an activated IC (i.e., the oracle) [19], [20], they can use Boolean satisfiability (SAT)-based attacks to recover the key. Several solutions have been proposed to thwart SAT-based attacks [21], [22]. For example, stripped-functionality logic locking (SFLL) extracts part of the functionality, which is hidden and restored upon the application of the correct key [22]. SFLL-HLS is the corresponding HLS-level extension [23]. However, complete protection is not guaranteed as attacks on SFLL have been reported when the "protected" functional inputs are at a certain Hamming distance from the key [24], [25]. Also, many SAT-resilient protections, such as SARLock and SFLL, can be broken even with oracle-less attacks [17]. Thus, the approaches for the different threat models are complementary and must be combined to obtain multilevel protection.

In this work, we aim at avoiding the attackers who can recover the circuit with modern oracle-less attacks. We base our techniques on the concept of indistinguishability; all Boolean functions generated by a locking key have the same probability of being the correct circuit. Thus, netlist-only attacks are not able to identify and rule out "incorrect" designs.

B. Article Contributions

ASSURE RTL obfuscation uses three techniques to obfuscate constants, arithmetic operations, and control branches. ASSURE provides the following contributions with respect to the state-of-the-art approaches.

1) The three ASSURE techniques are complete and provably secure for creating indistinguishable RTL designs with no limitations on the input descriptions to be protected.
2) ASSURE can provide multilevel security together with oracle-based protections (e.g., scan-chain isolation [26]).
3) ASSURE is a technology-independent tool that is fully compatible with existing EDA design flows and leaves complete control to the designer on the obfuscation process.

We describe our RTL-to-RTL translation framework in Section III, along with security proofs of obfuscations for constants, operations, and branches (see Section III-B). We also assess security against state-of-the-art oracle-less attacks (see Section IV-B) and evaluate the related overhead (see Section IV-D).

II. Threat Model: Untrusted Foundry

The state of the art in logic locking considers two broad categories of threat models: netlist-only and oracle-guided [8], [27]. In both settings, the attacker has access to a locked netlist, but in the latter, also to an unlocked IC (*oracle*) to analyze input–output relationships. The netlist-only model applies for an untrusted foundry that accesses the IC design for the first time. It also captures low-volume settings—e.g., the design of future defense systems with unique hardware requirements [28]—where the attacker would not reasonably be able to access a working copy of the IC. Consider, for instance, a fab-less defense contractor that outsources the fabrication of an IC to an untrusted foundry. The untrusted foundry has access to the layout files of the design and can reverse engineer a netlist and even extract the corresponding RTL [29]. 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 behavior (e.g., SAT attacks) cannot be applied and are therefore out-of-scope. However, the attacker can still rely on a range of netlist-only attacks, desynthesis [30], redundancy identification [17], and ML-guided structural and functional analysis [31], [32], for instance, to recover the key bits and 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. In the oracle-guided model, the attackers need to get an unlocked IC from the market—e.g., because of high-volume commercial fabrication—to analyze I/O relationships and apply the corresponding attacks. With our method, we thwart attacks that are successful for oracle-guided protections even without activated IC.

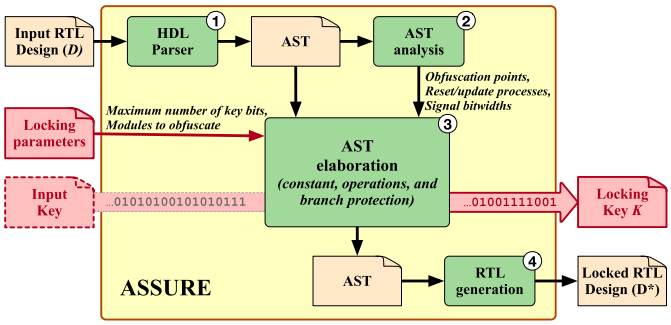


Fig. 2. Organization of ASSURE.

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^* together with a single locking 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 and those generated with commercial HLS tools. Even if logic locking is a hardware approach, obfuscating RTL code has analogies with program obfuscation to protect the software IP [14], [16]. In both cases, the designer aims at obfuscating the semantic information contained in the design rather than its structure [13]. 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.

1) Constants contain sensitive information in the computation (e.g., filter coefficients).
2) Operations determine functionality.
3) Branches define the execution flow (i.e., whose operations are executed under specific 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, and it fits with existing EDA flows and the 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 the 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 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.

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^*}
```

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 the 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). The resulting list `ObfElem` follows the visit order of the depth-first search. For each element, ASSURE computes the number of bits required for obfuscation (line 6) and check whether 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 [10] 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 to be added to $\mathcal{K}_r^*$ (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. This approach leaves full control to the designers that can explore tradeoffs by providing constraints on the number of bits and combining the depth-first analysis with the annotations to exclude elements from obfuscation.

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 [14]. 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 [30]. Consider an m-input n-output Boolean function $\mathcal{F}$: $X \rightarrow Y$, where $X \in \{0,1\}^m$ and $Y \in \{0,1\}^n$.

Definition: Obfuscation $\mathcal{O}(\mathcal{K}_r^*)$ transforms $\mathcal{F}$ into an $m+r$-input n-output function $\mathcal{L}_{\mathcal{K}_r^*}$ defined as

$$\mathcal{L}_{\mathcal{K}_r^*}(X, K) = \mathcal{O}\big(\mathcal{F}(X), \mathcal{K}_r^*\big) \tag{1}$$

where $\mathcal{L}_{\mathcal{K}_r^*}$: $X \times K \rightarrow Y$ and $K \in \{0,1\}^r$ are such that the following conditions hold.

1) $\mathcal{L}_{\mathcal{K}_r^*}(X, \mathcal{K}_r^*) = \mathcal{F}_{\mathcal{K}_r^*}(X) = \mathcal{F}(X)$.
2) $\mathcal{L}_{\mathcal{K}_r^*}(X, \mathcal{K}_r) = \mathcal{F}_{\mathcal{K}_r}(X) \neq \mathcal{F}(X)$ when $\mathcal{K}_r \neq \mathcal{K}_r^*$.

This definition shows that $\mathcal{C}_{\mathcal{K}_r^*}$ represents a family of Boolean functions $\{\mathcal{F}_{K_r}\}$ based on the generic r-bit key input $\mathcal{K}_r$. The functionality $\mathcal{F}(X)$ can be reobtained uniquely with the correct key $\mathcal{K}_r^*$. This is followed by a corollary about a characteristic of the family of Boolean functions $\mathcal{L}_{\mathcal{K}_r^*}$.

Theorem 1: For an obfuscated netlist $L_{\mathcal{K}_r^*}(X, K)$ created from $\mathcal{F}(X)$ with obfuscation $\mathcal{O}(\mathcal{K}_r^*)$, the unlocked functions $\mathcal{F}_{K_1}$ and $\mathcal{F}_{K_2}$ (for keys K_1 and K_2) relate as follows:

$$\mathcal{F}_{K_1} \neq \mathcal{F}_{K_2} \forall K_1, \quad K_2 \in K : K_1 \neq K_2. \tag{2}$$

Proof: Let us first consider case (i), $K_1 = \mathcal{K}_r^*$. Therefore, by the definition of RTL obfuscation scheme $\mathcal{O}$, $\mathcal{F}_{K_1} \neq \mathcal{F}_{K_2} \forall K_2 \in K, K_1 \neq K_2$. Now, for case (ii), $K_1 \neq \mathcal{K}_r^*$, there exists a locked netlist $\mathcal{L}_{\mathcal{K}_r^*}$ that locked $\mathcal{F}_{K_1}$ using K_1. Therefore, $\mathcal{F}_{K_2} = \mathcal{L}_{\mathcal{K}_r^*}(X, \mathcal{K}_2)$. By the definition of logic locking security, $\mathcal{F}_{K_2} \neq \mathcal{F}_{K_1} \forall K_2 \neq K_1$ in $\mathcal{L}_{\mathcal{K}_r^*}(X, \mathcal{K}_r)$. ■

We define $P[\mathcal{L}_{\mathcal{K}_r} | \mathcal{O}(\mathcal{F}(X), \mathcal{K}]$ as the probability of obtaining the locked design $\mathcal{L}_{\mathcal{K}_r}$, given that we locked the Boolean function $\mathcal{F}(X)$ applying $\mathcal{O}$ with $\mathcal{K}$. The RTL locking scheme $\mathcal{O}$ is secure under the netlist-only threat model.

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

Theorem 2: A logic locking scheme $\mathcal{O}$ 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:

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

This theorem states that any locking key $\mathcal{F}_{K_r}$ is equally probable to generate the locked netlist $\mathcal{L}_{\mathcal{K}_r'}$ generated by the locking scheme $\mathcal{O}$, creating a family of Boolean function $\mathcal{F}_{K_r}(X)$ all having the same probability to be the original Boolean function $\mathcal{F}(X)$. We show that all our obfuscations satisfy these two claims, providing a security guarantee of 2^r under the proposed threat model. This guarantee allows the designer to choose the parameter r to match the technology issues for storing the bits in the final IC. ASSURE will generate a locked design with the corresponding level of security.

1) Constant Obfuscation: This obfuscation removes selected constants and moves them into the locking key K, as shown in Fig. 3(a). 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 sensitive information (e.g., proprietary implementations of digital filters or cryptographic algorithms [33]) 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. Thus, we exclude constants related to reset processes and sensitivity values from 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_r^*. ASSURE constant obfuscation converts the r-bit constant into an r-bit key K_r^* as a lock $\mathcal{O}$ and uses it to lock the design

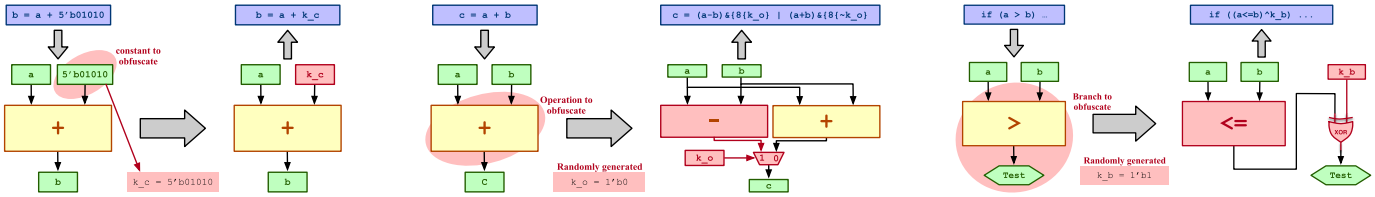


Fig. 3. Three ASSURE obfuscations. (a) Constant. (b) Operation. (c) Branch. Each obfuscation uses a portion of the key.

$\mathcal{L}_{\mathcal{K}_r^*}$. The obfuscated constant C_r is depicted as follows:

$$C_r = \mathcal{K}_r \tag{4}$$

where $C_r = C_r^*$ only when $\mathcal{K}_r = \mathcal{K}_r^* = C_r^*$.

Claim 1: Any unlocked constants 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:

$$P[C_r|K = \mathcal{K}_r^*] = P[C_r|K = K_r] \quad \forall K_r \neq K_r^*;\ K_r \in 2^r. \tag{5}$$

Proof: The probability of choosing K_r is uniform. Thus,

$$P[K = \mathcal{K}_r^*] = P[K = \mathcal{K}_r] \quad \forall K_r \neq K_r^*$$
$$\implies P[C_r^*] = P[C_r],\ C_r^* \neq C_r, \quad \forall C_r \in \{0,1\}^r.$$

■

Claims 1 and 2 jointly denote that the constant obfuscated by 2^r unique constants is 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 common evaluation platform (CEP) benchmarks [34]. 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 remains the same after obfuscation, and the obfuscated circuits are indistinguishable, making it difficult for the attacker to recover the constant.

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. 3(b). 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-level obfuscation [10], [35].

Example: Let us obfuscate RTL operation `c = a + b` with a dummy subtraction. We generate a key bit `k_o = 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

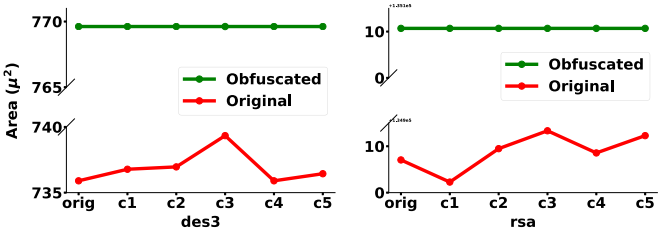


Fig. 4. Area of original and obfuscated variants of `DES3` and `RSA` when synthesized with different constants (c1–c5).

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 has the 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 $\mathcal{F} : X \to Y,\ X \in \{0,1\}^m$ and with r possible sites for operator obfuscation. ASSURE uses multiplexer-based locking $\mathcal{O}$ with an r-bit key $\mathcal{K}_r^*$ to lock the design and generate $\mathcal{L}_{\mathcal{K}_r^*}$

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

where $\mathcal{F}_{K_r^*}(X) = \mathcal{L}_{\mathcal{K}_r^*}(X, K = \mathcal{K}_r^*)$ ($\mathcal{K}_r^*$ is the 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:

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

Second, 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 the 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., $K_1 = 1001$ and $K_2 = 1100$, then $F_{K_1} \neq F_{K_2}$

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

Claim 1: Any pair of unlocked circuits $F_{K_r^1}$ and $F_{K_r^2}$ using r-bit keys K_r^1 and K_r^2 on multiplexer-based obfuscated circuit $\mathcal{L}_{\mathcal{K}_r^*}$ 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., the following condition holds true:

$$P[\mathcal{L}_{\mathcal{K}_r}|\mathcal{O}(\mathcal{F}(X), K_r^*)] = P[\mathcal{L}_{\mathcal{K}_r}|\mathcal{O}(\mathcal{F}(X), K_r^i)] \quad \forall K_r^i \neq K_r^*; F_{K_r^i} \neq F_{K_r^*}; i \in [1, 2^r].$$

Proof: The probability of choosing K_r is uniform. Therefore,

$P[K = K_r^*] = P[K = K_r^i], \forall K_r^i \neq K_r^*$
$\implies P[\mathcal{L}_{\mathcal{K}_r}(X, K = \mathcal{K}_r^*)] = P[\mathcal{L}_{\mathcal{K}_r}(X, K = \mathcal{K}_r^i)]$
$\implies P[\mathcal{F}_{\mathcal{K}_r^*}] = P[\mathcal{F}_{\mathcal{K}_r^i}] = (1/2^r)$. ■

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 that 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) is unique and can be reverse-engineered. On the contrary, the area overhead of synthesized circuits remains the same after obfuscation, and thus, the obfuscated circuits reveal 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 xor k_b`, as shown in Fig. 3(c). 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

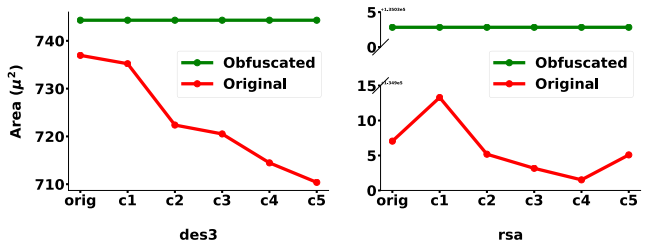


Fig. 5. Area of original and obfuscated variants of benchmarks `DES3` and `RSA` using different operators in the statement.

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)`$\wedge$`k_b`, which is equivalent to the original one only for the correct key bit. The attacker has no additional information to infer whether 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) $G(V, E)$ having $|V|$ 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 $V_{\text{CN}} = \{v_1, v_2, \ldots v_r\}$, $V_{\text{CN}} \subset V$ ($r = |V_{\text{CN}}|$). ASSURE applies XOR-based branch obfuscation to V_{CN} with r-bit key $\mathcal{K}_r^*$ as the logic locking scheme $\mathcal{O}$ and generates a locked design $\mathcal{C}_{\mathcal{K}_r^*}$. For example, if $V_{\text{CN}} = \{v_1, v_2, v_3, v_4\}$ and $K = 1101$, then $\mathcal{L}(V_{\text{CN}}) = \{\overline{v_1}, \overline{v_2}, v_3, \overline{v_4}\}$. The locked design, post branch obfuscation is illustrated as follows:

$$\begin{aligned}\mathcal{L}_{\mathcal{K}_r^*}(G(V, E), K) &= \mathcal{O}\big(G(V, E), K_r^*\big) \\ &= G\big(\mathcal{O}\big(V_{\text{CN}}, K_r^*\big) \cup (V \setminus V_{\text{CN}}), E\big) \\ &= G\big(\big(V_{\text{CN}} \oplus K_r^*\big) \cup (V \setminus V_{\text{CN}}), E\big)\end{aligned} \quad (9)$$

where $G(V, E) = \mathcal{L}_{\mathcal{K}_r^*}(G(V, E), K = \mathcal{K}_r^*)$.

Claim 1: The unlocked CFGs $\mathcal{L}_{\mathcal{K}_r^*}(G(V, E), K_1)$ and $\mathcal{L}_{\mathcal{K}_r^*}(G(V, E), K_2$ using r-bit keys K_1 and K_2, respectively, on the XOR-based encrypted CFG $\mathcal{L}_{\mathcal{K}_r^*}(G(V, E), K)$ are unique.

Proof: $\forall K_1 \neq K_2,\ K_1, K_2 \in \{0, 1\}^r$
$\implies K_1 \oplus V_{\text{CN}} \neq K_2 \oplus V_{\text{CN}} \implies V_{\text{CN}}^1 \neq V_{K_2}^2$.
$\implies G(V_{\text{CN}}^1, E) \neq G(V_{\text{CN}}^2, E)$. ■

Claim 2: The obfuscated CFG $\mathcal{L}_{\mathcal{K}_r^*}(G(V, E), K)$ can be generated from 2^r possible combination of condition statuses with equal probability, i.e., the following condition holds true:

$$P[\mathcal{L}_{\mathcal{K}_r'}(G(V, E), K)|G((V_{\text{CN}} \oplus K_r^*) \cup (V \setminus V_{\text{CN}}), E)] = P[\mathcal{L}_{\mathcal{K}_r'}(G(V, E), K)|G((V_{\text{CN}} \oplus K*_r) \cup (V \setminus V_{\text{CN}}), E)] \quad \forall K_r \neq K_r^*; V_{\text{CN}}^r \neq V_{\text{CN}}. \quad (10)$$

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	127,667	8,502	506	21,812	136,493	42,854.69	136.75
	DES3	11	4	3	775	898	2,076	135	128	368	2,448	736.96	192.28
	DFT	211	447	151	132	8,697	118,201	38,521	9,552	41,320	158,807	81,865.94	336.72
	FIR	5	10	24	0	344	820	439	49	225	1,704	1,129.36	377.76
	IDFT	211	447	151	132	8,697	118,154	38,525	9,576	41,305	158,722	81,821.90	333.59
	IIR	5	19	43	0	651	1,378	648	72	367	2,621	1,679.72	464.82
	MD5	2	150	50	1	4,533	4,682	269	168	923	5,756	1,840.15	791.53
	RSA	15	243	35	13	1,942	222,026	57,987	21,808	66,088	280,222	134,907.05	386.55
	SHA256	3	159	36	2	4,992	5,574	1,040	243	1,024	7,532	3,201.07	440.67
IWLS	MEM_CTRL	27	492	442	160	2,096	4,007	1,051	120	1,136	5,183	2,373.35	260.72
	SASC	3	35	27	17	126	367	116	0	125	500	238.24	84.40
	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	34,783	10,545	2,195	12,021	45,441	22,453.76	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	54,614	17,052	4,921	19,228	71,766	36,095.90	224.67
OpenROAD	ARIANE_ID	4	3,498	385	723	4,606	1,993	378	96	559	2,615	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	12,161	1,864	978	2,965	14,379	5,758.84	538.10

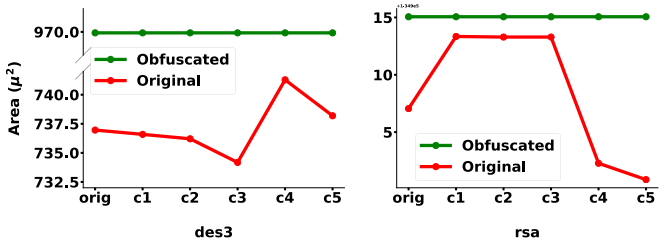


Fig. 6. Area of original and obfuscated variants of benchmarks DES3 and RSA in case of different CFG flows.

Proof: The probability of choosing K_r is uniform. Thus,
$P[K = \mathcal{K}_r^*] = P[K = \mathcal{K}_r], \forall \mathcal{K}_r \neq \mathcal{K}_r^*, \mathcal{K}_r \in 2^r$
$\implies P[(V_{\text{CN}} \oplus \mathcal{K}_r^*) \oplus \mathcal{K}_r^*] = P[(V_{\text{CN}} \oplus \mathcal{K}_r^*) \oplus \mathcal{K}_r]$
$\implies P[V_{\text{CN}}] = P[V_{\text{CN}}^r], V_{\text{CN}} \neq V_{\text{CN}}^r,$
$V_{\text{CN}}^r = \{p_1, p_2, \ldots, p_i, \ldots, p_r\}$, where $p_i \in \{v_i, \overline{v_i}\}$. ■

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 remains the same after obfuscation, indicating that the obfuscated circuits reveal no information about the control flow of the circuit.

IV. EXPERIMENTAL VALIDATION OF ASSURE

A. Experimental Setup

We implemented ASSURE as a Verilog→Verilog tool that leverages Pyverilog [36], 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 then reproduces the output Verilog description ready for logic synthesis.

We used ASSURE to protect several Verilog designs from different sources[2]: the MIT-LL CEP platform [34], the OpenROAD project [37], and the OpenCores repository [38]. Four CEP benchmarks (DCT, IDCT, FIR, and IIR) are created with Spiral, a hardware generator [39]). Table I shows the characteristics of these benchmarks in terms of a number of hardware modules, constants, operations, and branches. These data also characterize the functionality that needs obfuscation. The benchmarks are much larger than those used by the gate-level logic locking experiments by the community [9]. Different from [10], ASSURE does not require any modifications to synthesis tools and applies to preexisting industrial designs, processing the Verilog RTL descriptions with no modifications.

We analyzed the ASSURE in terms of security (see Sections IV-B and IV-C) and overhead (see Section IV-D). 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 the obfuscation of the DFT benchmark, where all three obfuscations are applied using 2175 bits for obfuscation (25% of 8697) as follows: 38 for operations (25% of 151), 33 for branches (25% of 132), and the rest (2104) for constants.

B. Correctness and Key Effects

We first apply formal verification on the locked design against the unprotected design with a twofold goal. 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 every single

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

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

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 no other key can activate the same IC, i.e., all other circuits ($K_r \neq K_r^*$) are not exact copies of the original designs. In this experiment, we also aim at quantifying how the obfuscation techniques affect the IC functionality when the attacker provides incorrect keys. With formal verification, we focus on IC functionality rather than IC results. We compute the verification failure metric as

$$F = \frac{1}{K} \cdot \sum_{i=1}^{K} \frac{n(\text{Failing Points})_i}{n(\text{Total Points})}. \tag{11}$$

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.

1) Correctness: We apply ASSURE several times, each time with a random key to obfuscate operations and branches (constants are always extracted in the same way. We formally verified these designs against the original ones. In all experiments, ASSURE generates circuits that match the original design when using the correct key.

2) Key Effect: Given a design obfuscated with an r-bit key, we performed r experiments where, in each of them, we flipped 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. 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 ($\sim 10^{-5}$) since they 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, obfuscation is closer to the output and more internal points match.

This experiment allowed us to identify design practices that lead to inefficient obfuscations or even collisions. In `DFT`, one-bit signals were initialized with integer values 0/1. Verilog allows this syntax and 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. Thus, we extended the ASSURE AST analysis to match the constant sizes with those of the target signals.

TABLE II

SECURITY ASSESSMENT OF ASSURE OBFUSCATION AGAINST REDUNDANCY ATTACKS [17] (WITHOUT ORACLE) AND KC2 ATTACKS [40] (WITH ORACLE). FAILED DENOTES ATTACK FAILURE DUE TO NONEXISTENCE OF UNTESTABLE FAULTS FOR REDUNDANCY ATTACKS AND DUE TO UNSOLVABLE CONSTRAINTS OR INCORRECT KEY GENERATION FOR KC2 ATTACKS. TIMEOUT DENOTES TOOL TERMINATION AFTER 96 h WITHOUT RETURNING THE KEY. CFG1, CFG2, CFG3, AND CFG4 CORRESPOND TO 25%, 50%, 75%, AND 100% OF MAXIMUM POSSIBLE KEY-BIT OBFUSCATION, RESPECTIVELY. (X/Y) IN REDUNDANCY ATTACK INDICATE X KEY BITS ARE CORRECT OUT OF Y KEY BITS RECOVERED BY THE REDUNDANCY ATTACK

Bench mark	Obf. Type	Attack with oracle access?	Obfuscation configuration											
			CFG1			CFG2			CFG3			CFG4		
			Key (bits)	Recovered (bits)	Time (s)	Key (bits)	Recovered (bits)	Time (s)	Key (bits)	Recovered (bits)	Time (s)	Key (bits)	Recovered (bits)	Time (s)
DES3	All	no	225	20/34	5,655	450	31/54	20,860	675	0	timeout	900	0	timeout
		yes		225	13,447		450	16,216		0	failed		0	timeout
	Constant	no	30	0/8	264	60	0/8	968	90	0/10	1,456	120	0/10	2,575
		yes		30	2,324		60	5,398		0	failed		120	8,476
FIR	All	no	86	4/32	3,269	164	7/45	26,045	250	12/67	39,025	336	0	timeout
		yes		0	1,372		0	failed		0	5,665		0	timeout
	Constant	no	80	0/25	2,989	152	0/26	22,697	232	0/52	33,156	312	0	timeout
		yes		0	1,189		0	failed		0	5,145		0	timeout
MD5	All	no	1,135	0	timeout	2,267	0	timeout	3,401	0	timeout	4,533	0	timeout
		yes		0	failed		0	timeout		0	timeout		0	timeout
	Constant	no	1,121	0	timeout	2,241	0	timeout	3,362	0	timeout	4,482	0	timeout
		yes		0	failed		0	timeout		0	timeout		0	timeout
SHA256	All	no	1,250	0	timeout	2,496	0	timeout	3,745	0	timeout	4,992	0	timeout
		yes		0	failed		0	failed		0	timeout		0	timeout
	Constant	no	1,239	0	timeout	2,477	0	timeout	3,716	0	timeout	4,954	0	timeout
		yes		0	failed		0	failed		0	timeout		0	timeout
SS_PCM	All	no	7	0/4	2	13	0/4	3	18	1/5	5	24	1/5	7
		yes		7	843		13	170		18	1,308		0	6,052
	Constant	no	3	0/0	2	6	0/0	2	8	0/0	3	11	0/0	5
		yes		3	289		6	310		8	784		0	1897
GCD	All	no	11	3/11	8	23	5/15	8	34	7/17	12	47	9/16	14
		yes		0	8		0	15		0	15		0	21
	Constant	no	7	0/0	6	15	0/4	7	22	0/8	11	31	0/8	14
		yes		0	7		0	7		0	14		0	19
USB_ PHY	All	no	57	15/21	17	112	0	failed	163	34/75	105	223	47/86	184
		yes		0	521		0	548		0	898		0	360
	Constant	no	30	0/0	14	60	0	failed	89	0/5	97	119	0/10	152
		yes		0	510		0	522		0	524		0	347

C. Resilience Against Locking Attacks

We outlined provable security guarantees that n obfuscation bits induce 2^n RTL designs with uniform probability. We now discuss resilience to known locking attacks.

1) Resynthesis Attacks: Massad *et al.* [30] showed that greedy heuristics can recover the key of an obfuscated gate-level netlist. Performing resynthesis with an incorrect key may trigger additional optimizations that produce large redundancy in the circuit. Similarly, Li and Orailoglu [17] proposed an attack using concepts from VLSI testing. Incorrect key results in large logic redundancy and most of the stuck-at faults become untestable. A correctly unlocked circuit, however, has high testability. ASSURE obfuscates RTL design before synthesis. Since the obfuscated RTL is equally likely to be generated from 2^n designs, 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 the same range (see Figs. 4–6).

2) ML-Guided Attacks: Chakraborty *et al.* [31], [32] proposed oracle-less attacks on logic obfuscation based on the idea that obfuscation techniques insert XOR/XNOR gates that leave structural traces. The key gates are inserted before synthesis with known technology library and synthesis process (algorithms and tools). Since the effect of logic optimizations is local and the optimization rules are deterministic, one can recover the original function by launching an ML-guided removal attack on the obfuscated RTL design. In ASSURE, the obfuscation logic does not depend solely on the insertion of XOR/XNOR gates. For example, in branch obfuscation, we perform also logic inversion of the condition instead of simply adding a XOR gate followed by a NOT when the corresponding key bit is 1. Recovering the original RTL from obfuscated RTL is hard (see claim 2 of ASSURE branch obfuscation proof in Section III-B3). Also, recovering extracted constants from an obfuscated design is impossible since the obfuscated circuit does not contain any information on the constant value.

3) Redundancy and KC2 Attacks: We analyze the strength of ASSURE's obfuscation by running oracle-less redundancy attacks [17]. Redundancy attacks decipher the key bits by identifying redundant lines in the synthesized netlist with incorrect key bits. KC2 is an improved version of SAT-based attacks incrementally unrolling a sequential circuit to recover the key. Even if ASSURE is not designed to protect against oracle-guided attacks, we evaluated its performance on KC2 [40], a popular oracle-guided attack. We have run both attacks with a timeout of 96 h and 50 GB of memory for each attack run. Table II summarizes the results of both attacks on selected ASSURE obfuscated designs. In particular, we apply the attacks to benchmarks that we can safely convert into the format required by the attack tools. We perform the attacks after applying all obfuscations (`ALL`) or after applying only constant obfuscation (`Constant`). Constant obfuscation successfully thwarts all redundancy attacks, showing that this is the most powerful obfuscation. Indeed, on benchmarks such as `DES3`, `FIR`, `SS_PCM`, and `USB_PHY`, redundancy attacks recovered some key bits. These results indicate that combining different obfuscation techniques is not 100% secure compared with stand-alone obfuscation. Even if we focused on the

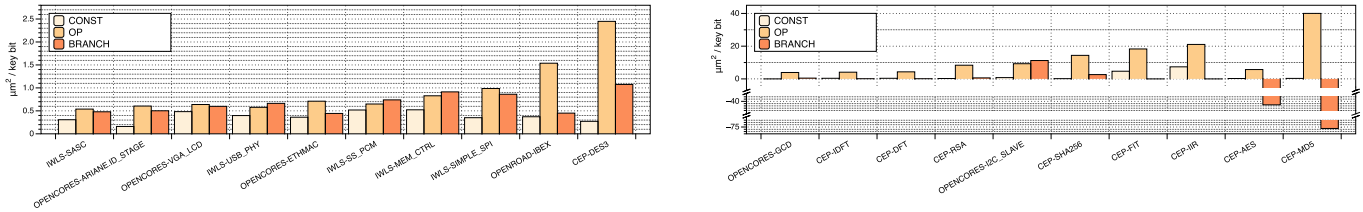


Fig. 8. Area overhead per key bit for ASSURE obfuscation. Benchmarks are presented in increasing order of total overhead.

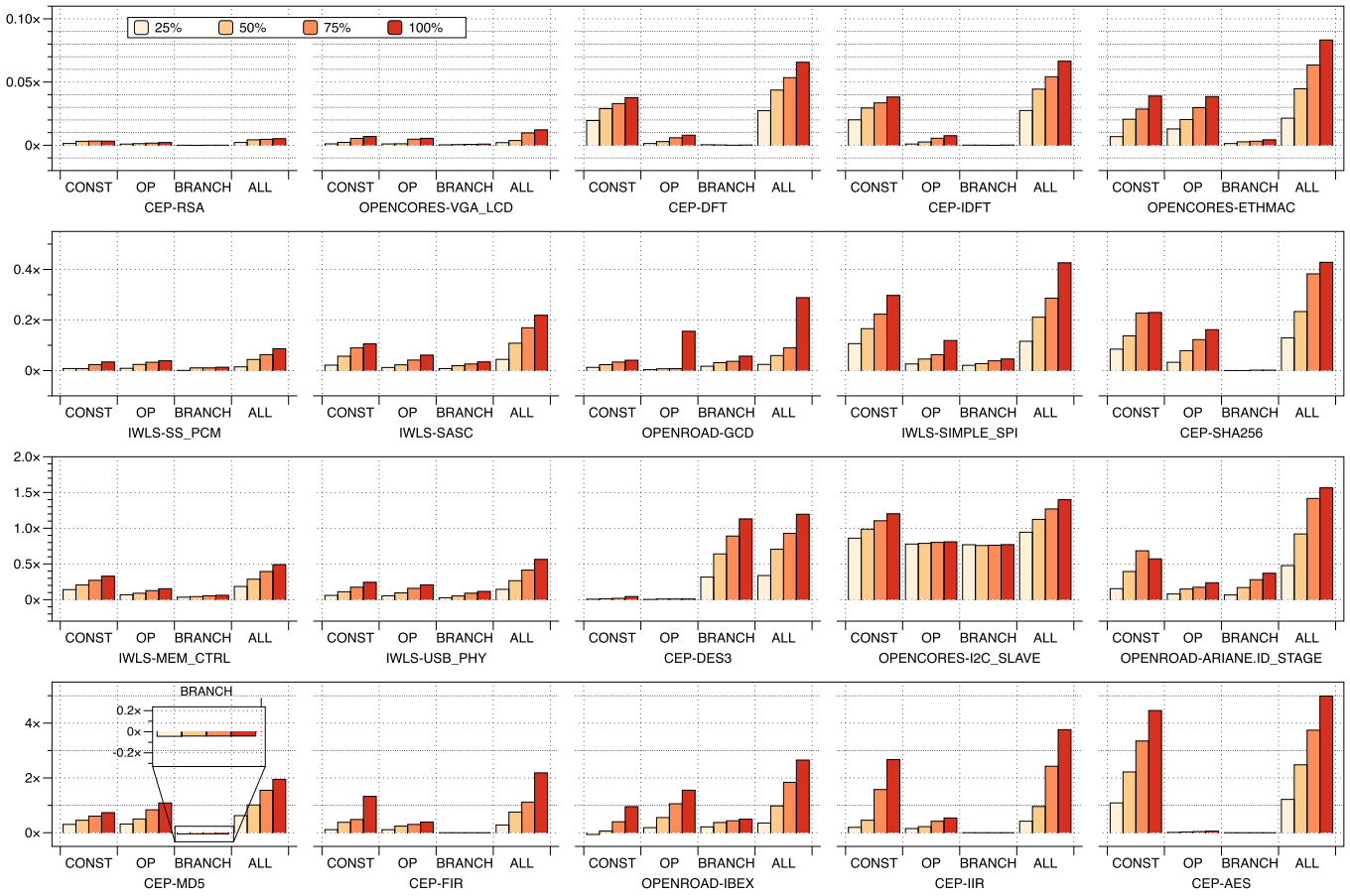


Fig. 9. Area overhead for ASSURE obfuscation. Benchmarks are presented in increasing order of total overhead.

netlist-only threat model, it is interesting to evaluate the effects of oracle-guided attacks. KC2 attacks were able to recover the correct keys only for `DES3` and `SS_PCM` benchmarks. In all other cases, KC2 claimed to recover certain key bits. However, the equivalence checking performed by ABC within the tool showed that the functionality unlocked with those bits is not equivalent with the original one, i.e., the key is incorrect.

D. Synthesis Overhead

We did logic synthesis using the Synopsys Design Compiler J-2018.04-SP5 targeting the Nangate 15-nm ASIC technology at standard operating conditions (25 °C). 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.

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. 8 reports the area overhead of all obfuscations with respect to the original designs. The three techniques are independent, and thus, `ALL` results are the aggregate of the three techniques. Constant obfuscation produces an average overhead in the range from 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 such as constant propagation. In `AES`, all S-Box modules are optimized as logic in the original circuit. This optimization is not possible anymore when the constants are provided as inputs. However, we showed that this obfuscation provides maximum protection since the constants are

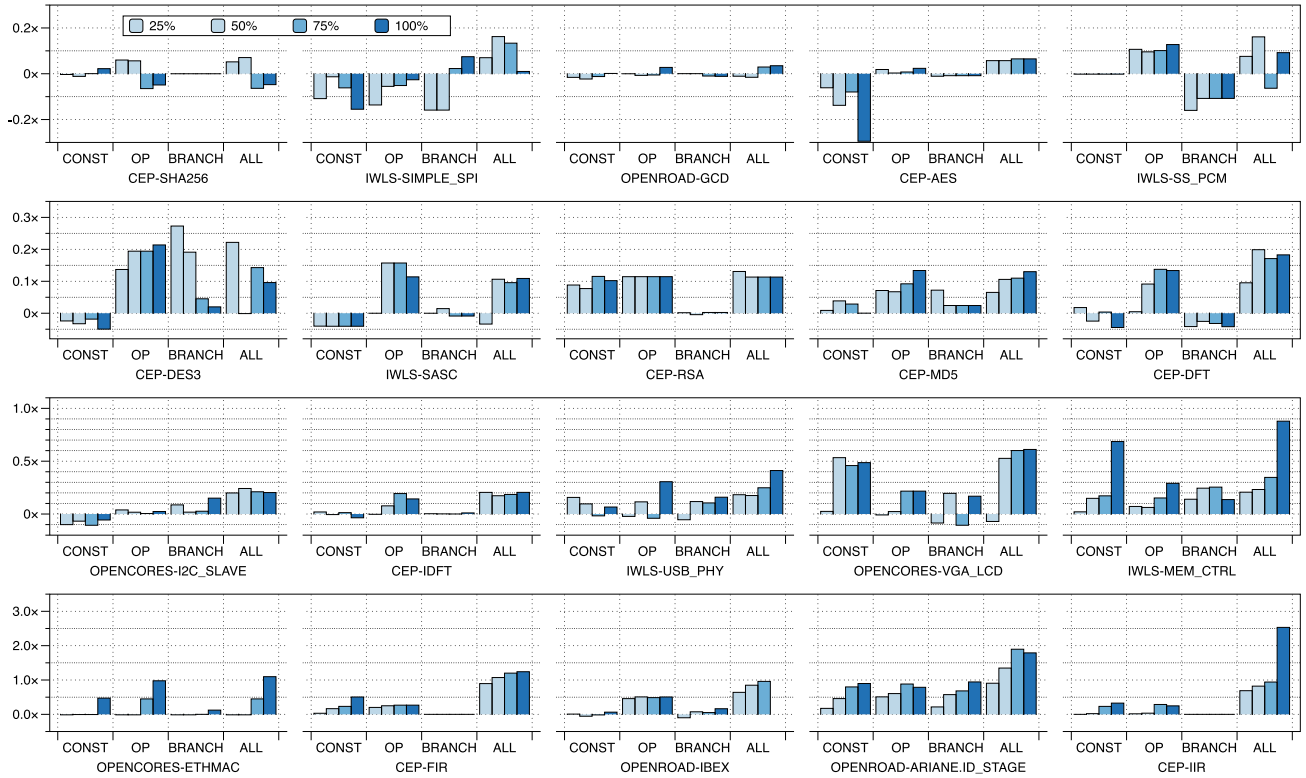


Fig. 10. Timing overhead for ASSURE obfuscation. Benchmarks are presented in increasing order of total overhead.

semantically removed from the circuit. The average operation obfuscation overhead is in the range from 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 from 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. Thus, 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. 9. In most cases, operation obfuscation has the largest impact, followed by branches and then constants. This impact is larger for data-intensive benchmarks, such as 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 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 observed minimal variants with different locking keys.

To conclude, the area overhead is related to the design characteristics and 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 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 the critical path) designs. In most benchmarks, the timing overhead is <20%. Constants have a positive impact on the overhead (see `AES` and `DES3`). The obfuscated designs can generally achieve the same performance as the original ones, limiting the impact on the IC design flow.

V. Conclusion and Discussion

We presented ASSURE, an RTL locking framework against an untrusted foundry that has no access to an unlocked functional chip. ASSURE operates on the Verilog RTL description and is fully compatible with industrial EDA flows. We discuss the major contributions in the form of Q&A.

Which Threat Model Are You Considering? How Is It Relevant for My Design?: We consider the netlist-only threat model where the attacker has no access to an activated chip. This model is relevant, especially for an untrusted foundry with low-volume IC production.

Why Should I Use an RTL Approach Instead of Existing Gate-Level Techniques?: 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. Most of the semantic information (e.g., constants) cannot be protected at the gate level because synthesis tools embed it into the netlist.

Is ASSURE Secure?: In our experimental analysis with formal verification and logic synthesis EDA tools, we show that the circuits can be unlocked only with the correct key and obfuscating the design closer to the inputs induces more verification failures. Also, ASSURE can thwart oracle-less attacks that can recover key bits even in case of SAT-resilient protections, showing that the two approaches must be combined.

What Is the Overhead?: 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 prevents logic optimizations, such as constant propagation, while operation obfuscation has the largest overhead per key bit. 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. The designers can use these guidelines to apply obfuscation on their design.

ACKNOWLEDGMENT

The authors would like to thank Benjamin Tan and Jitendra Bhandari, New York University (NYU), for their support in implementing locking attacks.

REFERENCES

[1] S. W. Jones, "Technology and cost trends at advanced nodes," IC Knowl. LLC, Georgetown, MA, USA, Tech. Rep., 2019.

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

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

[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," *Proc. IEEE*, vol. 102, no. 8, pp. 1207–1228, Aug. 2014.

[5] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L.-C. Wang, "Challenges and trends in modern SoC design verification," *IEEE Des. Test*, vol. 34, no. 5, pp. 7–22, Oct. 2017.

[6] J. Rajendran, O. Sinanoglu, and R. Karri, "Is split manufacturing secure?" in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2013, pp. 1259–1264.

[7] A. T. Abdel-Hamid, S. Tahar, and E. M. Aboulhamid, "IP watermarking techniques: Survey and comparison," in *Proc. 3rd IEEE Int. Workshop Syst.-Chip Real-Time Appl.*, Jul. 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 Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 6, pp. 1–36, Nov. 2019.

[9] B. Tan *et al.*, "Benchmarking at the frontier of hardware security: Lessons from logic locking," 2020, *arXiv:2006.06806*. [Online]. Available: http://arxiv.org/abs/2006.06806

[10] C. Pilato, F. Regazzoni, R. Karri, and S. Garg, "TAO: Techniques for algorithm-level obfuscation during high-level synthesis," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018, pp. 1–6.

[11] 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, Switzerland: Springer, 2020, pp. 172–192.

[12] Y. Lao and K. K. Parhi, "Obfuscating DSP circuits via high-level transformations," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 5, pp. 819–830, May 2015.

[13] R. S. Chakraborty and S. Bhunia, "RTL hardware IP protection using key-based control and data flow obfuscation," in *Proc. 23rd Int. Conf. VLSI Design*, Jan. 2010, pp. 405–410.

[14] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Dept. Comput. Sci., Univ. Auckland, Auckland, New Zealand, Tech. Rep. 148, 1997.

[15] C. K. Behera and D. L. Bhaskari, "Different obfuscation techniques for code protection," in *Proc. Int. Conf. Eco-Friendly Comput. Commun. Syst.*, vol. 70, 2015, pp. 757–763.

[16] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu, "On secure and usable program obfuscation: A survey," 2017, *arXiv:1710.01139*. [Online]. Available: http://arxiv.org/abs/1710.01139

[17] L. Li and A. Orailoglu, "Piercing logic locking keys through redundancy identification," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 540–545.

[18] D. Sisejkovic, F. Merchant, L. M. Reimann, H. Srivastava, A. Hallawa, and R. Leupers, "Challenging the security of logic locking schemes in the era of deep learning: A neuroevolutionary approach," *ACM J. Emerg. Technol. Comput. Syst.*, 2020.

[19] S. Amir *et al.*, "Development and evaluation of hardware obfuscation benchmarks," *J. Hardw. Syst. Secur.*, vol. 2, no. 2, pp. 142–161, Jun. 2018.

[20] Y. Shen, Y. Li, A. Rezaei, S. Kong, D. Dlott, and H. Zhou, "BeSAT: Behavioral SAT-based attack on cyclic logic encryption," in *Proc. 24th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2019, pp. 657–662.

[21] Y. Xie and A. Srivastava, "Anti-SAT: Mitigating SAT attack on logic locking," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 2, pp. 199–207, Feb. 2019.

[22] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. J. Rajendran, and O. Sinanoglu, "Provably-secure logic locking: From theory to practice," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1601–1618.

[23] M. Yasin, C. Zhao, and J. J. Rajendran, "SFLL-HLS: Stripped-functionality logic locking meets high-level synthesis," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2019, pp. 1–4.

[24] F. Yang, M. Tang, and O. Sinanoglu, "Stripped functionality logic locking with Hamming distance-based restore unit (SFLL-hd)–unlocked," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 10, pp. 2778–2786, Oct. 2019.

[25] D. Sirone and P. Subramanyan, "Functional analysis attacks on logic locking," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 1–6.

[26] N. Limaye, E. Kalligeros, N. Karousos, I. G. Karybali, and O. Sinanoglu, "Thwarting all logic locking attacks: Dishonest oracle with truly random logic locking," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, Oct. 6, 2020, doi: 10.1109/TCAD.2020.3029133.

[27] K. Shamsi, D. Z. Pan, and Y. Jin, "On the impossibility of approximation-resilient circuit locking," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, May 2019, pp. 161–170.

[28] Defense Science Board Task Force. (2005). *Report on High Performance Microchip Supply*. [Online]. Available: https://www.hsdl.org/?abstract&did=454591

[29] J. Rajendran, A. Ali, O. Sinanoglu, and R. Karri, "Belling the CAD: Toward security-centric electronic system design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 11, pp. 1756–1769, Nov. 2015.

[30] M. El Massad, J. Zhang, S. Garg, and M. V. Tripunitara, "Logic locking for secure outsourced chip fabrication: A new attack and provably secure defense mechanism," 2017, *arXiv:1703.10187*. [Online]. Available: http://arxiv.org/abs/1703.10187

[31] P. Chakraborty, J. Cruz, and S. Bhunia, "SAIL: Machine learning guided structural analysis attack on hardware obfuscation," in *Proc. Asian Hardw. Oriented Secur. Trust Symp. (AsianHOST)*, Dec. 2018, pp. 56–61.

[32] P. Chakraborty, J. Cruz, and S. Bhunia, "SURF: Joint structural functional attack on logic locking," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, May 2019, pp. 181–190.

[33] S. Picek, L. Batina, D. Jakobović, B. Ege, and M. Golub, "S-box, SET, Match: A toolbox for S-box analysis," in *Information Security Theory and Practice. Securing the Internet of Things*. Berlin, Germany: Springer, 2014, pp. 140–149.

[34] MIT Lincoln Laboratory. *Common Evaluation Platform (CEP)*. Accessed: Apr. 10, 2021. [Online]. Available: https://github.com/mit-ll/CEP

[35] H. Badier, J.-C.-L. Lann, P. Coussy, and G. Gogniat, "Transient key-based obfuscation for HLS in an untrusted cloud environment," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 1118–1123.

[36] S. Takamaeda-Yamazaki, "Pyverilog: A Python-based hardware design processing toolkit for Verilog HDL," in *Proc. Int. Symp. Appl. Reconfigurable Comput. (ARC)*, Apr. 2015, pp. 451–460.
[37] T. Ajayi *et al.*, "Toward an open-source digital flow: First learnings from the OpenROAD project," in *Proc. 56th Annu. Design Autom. Conf.*, Jun. 2019, pp. 1–4.
[38] Oliscience *OpenCores Repository*. Accessed: Apr. 10, 2021. [Online]. Available: https://opencores.org/
[39] M. Püschel, F. Franchetti, and Y. Voronenko, "Spiral," in *Encyclopedia of Parallel Computing*. Springer, 2011.
[40] K. Shamsi, M. Li, D. Z. Pan, and Y. Jin, "KC2: Key-condition crunching for fast sequential circuit deobfuscation," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 534–539.

Christian Pilato (Senior Member, IEEE) received the Ph.D. degree in information technology from the Politecnico di Milano, Milan, Italy, in 2011.

He was a Postdoctoral Research Scientist at Columbia University, New York, NY, USA, from 2013 to 2016, and the Università della Svizzera italiana, Lugano, Switzerland, from 2016 to 2018. He was a Visiting Researcher at New York University, New York, TU Delft, Delft, The Netherlands, and the Chalmers University of Technology, Gothenburg, Sweden. He is currently a tenure-track Assistant Professor at the Politecnico di Milano. His research interests include high-level synthesis, reconfigurable systems, and system-on-chip architectures, with emphasis on memory and security aspects.

Dr. Pilato is a Senior Member of the Association for Computing Machinery and a member of HiPEAC. He served as the Program Chair for EUC 2014 and is also serving in the program committee of many conferences on EDA, CAD, embedded systems, and reconfigurable architectures (DAC, ICCAD, DATE, CASES, FPL, ICCD, and so on).

Animesh Basak Chowdhury (Graduate Student Member, IEEE) received the M.S. degree in computer science from the Indian Statistical Institute, Chennai, India, in 2016. He is currently working toward the Ph.D. degree at the NYU Center for Cybersecurity, New York University, New York, NY, USA.

Prior to joining the Ph.D. program, he spent three years as a Researcher at the Tata Research Development and Design Centre (TRDDC), Pune, India, where he was primarily working in the area of formal verification and security testing. His research interests include secure electronics design automation (EDA), machine learning, and system-on-chip (SoC) security.

Mr. Chowdhury has received several awards and recognition in International Software Verification and Testing Competitions (SV-COMP, TEST COMP, and RERS-Challenge).

Donatella Sciuto (Fellow, IEEE) received the Laurea (M.S.) degree in electronic engineering from the Politecnico di Milano, Milan, Italy, in 1984, the Ph.D. degree in electrical and computer engineering from the University of Colorado at Boulder, Boulder, CO, USA, in 1988, and the M.B.A. degree from Bocconi University, Milan, in 1992.

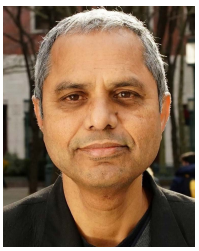
She is currently the Executive Vice-Rector of the Politecnico di Milano, where she is also a Full Professor of Computer Science and Engineering. More recently, she has been involved in managing and developing research projects in the area of smart cities and in the application of new ICT technologies to different application fields. She has published over 300 scientific articles. Her main research interests include the methodologies for the design of embedded systems and multicore systems considering performance, power, and security metrics.

Dr. Sciuto is a Fellow of IEEE for her contributions in embedded system design. She has served as the Vice-President for Finance and then President for the IEEE Council of Electronic Design Automation from 2009 to 2013. She serves in different capacities in IEEE Awards Committees and scientific boards of IEEE journals and conferences.

Siddharth Garg (Member, IEEE) received the B.Tech. degree in electrical engineering from IIT Madras, Chennai, India, in 2004, and the Ph.D. degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, USA, in 2009.

He was an Assistant Professor at the University of Waterloo, Waterloo, ON, Canada, from 2010 to 2014. In Fall 2014, he joined New York University (NYU), New York, NY, USA, as an Assistant Professor. His general research interests are in computer engineering and, more particularly, in secure, reliable, and energy-efficient computing.

Dr. Garg received the NSF CAREER Award in 2015, paper awards at the IEEE Symposium on Security and Privacy (S&P) 2016, the USENIX Security Symposium 2013, the Semiconductor Research Consortium TECHCON in 2010, and the International Symposium on Quality in Electronic Design (ISQED) in 2009, the Angel G. Jordan Award from the Department of Electrical and Computer Engineering, Carnegie Mellon University, for outstanding thesis contributions and service to the community. In 2016, he was listed in Popular Science Magazine's annual list of "Brilliant 10" researchers. He serves on the technical program committee of several top conferences in the areas of computer engineering and computer hardware. He has served as a reviewer for several IEEE and ACM journals.



Ramesh Karri (Fellow, IEEE) received the B.E. degree in electrical and computer engineering from Andhra University, Visakhapatnam, India, in 1985, and the Ph.D. degree in computer science and engineering from the University of California at San Diego, La Jolla, CA, USA, in 1993.

He is currently a Professor of Electrical and Computer Engineering at New York University, New York, NY, USA. He co-directs the NYU Center for Cyber Security. He founded the Embedded Systems Challenge, the annual red team blue team event. He co-founded Trust-Hub. His research and education activities in hardware cybersecurity include trustworthy integrated circuits (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; and additive manufacturing security. He has published over 250 articles in leading journals and conference proceedings.

Dr. Karri is a Fellow of the IEEE for leadership and contributions to Trustworthy Hardware. His work on hardware cybersecurity received the best paper nominations at ICCD 2015 and DFTS 2015 and awards at ACM TODAES 2018, ITC 2014, CCS 2013, DFTS 2013, and VLSI Design 2012. He received the Humboldt Fellowship and the NSF CAREER Award. He serve(d)s on the Editorial Board of IEEE and ACM Transactions, including IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, TODAES, ESL, D&T, and JETC. He served on the Executive Committee of the IEEE/ACM DAC leading the SecurityDAC initiative from 2014 to 2017. He served as the program/general chair for conferences and serves on program committees. He is the Editor-in-Chief of *ACM Journal on Emerging Technologies in Computing Systems* (JETC). He was an IEEE Computer Society Distinguished Visitor from 2013 to 2015.