High-Level Synthesis Techniques for Algorithm-Level Obfuscation

Christian Pilato, Donatella Sciuto, Francesco Regazzoni, Siddharth Garg, Ramesh Karri

Abstract Intellectual Property (IP) theft is one of the major concerns for the economy of semiconductor companies, costing billions of dollars every year. To make unauthorized IP copies, an attacker must reverse engineer and replicate the functionality of the given chip design. While existing IP protection techniques aim at manipulating HDL descriptions to thwart the reverse engineering process, they focus on the given implementation and fail in hiding all details of the the chip functionality. We propose a comprehensive solution to address this problem during high-level synthesis in order to apply obfuscation at the algorithm level. Our solution includes several key-based transformations that are applied during component generation to make reverse engineering harder during chip fabrication, while the key is later provided to the circuit to unlock the functionality. We show that our method is a promising approach to obfuscate large-scale designs despite the obfuscation overhead.

1 Introduction

The design flow for producing an Integrated Circuit (IC) is shown in Fig. 1. It is composed of several phases, ranging from the design of the components to the phys-

Christian Pilato
Politecnico di Milano, Italy e-mail: christian.pilato@polimi.it

Donatella Sciuto
Politecnico di Milano, Italy e-mail: donatella.sciuto@polimi.it

Francesco Regazzoni
ALaRI Institute, Università della Svizzera italiana, Switzerland e-mail: regazzoni@alari.ch

Siddharth Garg
New York University, USA e-mail: siddharth.garg@nyu.edu

Ramesh Karri
New York University, USA e-mail: rkarry@nyu.edu

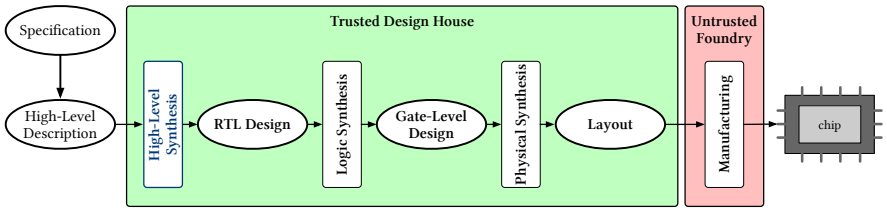

Fig. 1 IC design flow.

ical implementation on the target technology and the fabrication of the device on silicon. While the design phases only require access to commercial CAD tools, fabrication requires expensive infrastructures to function. For example, TSMC expect to invest more than 20 billions of dollars for 3nm foundries [23]. Many semiconductor companies cannot afford the increasing cost of IC manufacturing [10]. As the technology scales, more and more companies are becoming *fab-less*, outsourcing the IC fabrication to third-party foundries [11]. While this process allows for a cost reduction, it creates security threats in the semiconductor supply chain: an attacker that has access the design files can reverse engineer the functionality and steal the Intellectual Property (IP) [9]. Since these ICs often implement proprietary optimized designs, this malicious process can cause significant economic harm to the design houses. While IC watermarking is able to determine the real ownership of an IC during litigation, this is a passive method that requires to identify the illegal copy and enter into a legal dispute [1].

Semiconductor design houses are thus showing an increasing interest for anti-reverse engineering techniques for untrusted foundries. For example, *split manufacturing* divides computing resources and interconnections, with the two parts fabricated in different foundries. This process is based on the assumption that collusion between the two foundries is unlikely. However, designing for split manufacturing is complex and expensive. *Obfuscation* and *logic locking* have been extensively investigated for this purpose as well [29]. The designer adds additional inputs and modules to the design to hide the correct functionality (obfuscation), while a *locking key* (unknown to the foundry and written later in a tamper-proof memory) activates the IC (logic locking). These methods are usually applied on the gate-level netlist [25, 30]. With the increasing complexity of ICs, designers are migrating to *high-level synthesis* (HLS) to automate the design process [17]. While security features can be applied at any design steps, more robust solutions can be applied in the early stages, i.e., during HLS [18, 20, 22]. For example, the Stripped-Functionality Logic Locking approach (SFLL) [33] has been recently extended to HLS [34]. However, a holistic solution that brings together HLS and obfuscation is still missing.

In this chapter, we discuss a possible approach to rise the abstraction level of register transfer level (RTL) obfuscation by embracing a security-aware HLS flow to generate obfuscated designs by construction. We propose an approach based on **algorithm-level obfuscation**, which aims at developing anti-reverse engineering techniques based on the characteristics of the algorithm during the different HLS

steps. The approach we present starts from a high-level description of the functionality in C language, and use HLS methods to produce the corresponding obfuscated RTL description. This is achieved by obfuscating the HLS results or the generated RTL description. For doing this, HLS algorithms are extended to obfuscate the most sensitive details of an algorithm. After compiler analysis, the information that comes from the specification (e.g., constant values, loop bounds) and the information generated by HLS (e.g., control states, used and unused datapath resources, execution latency) are obfuscated. It is possible to obfuscate complex functions as part of a comprehensive HLS-based obfuscation design flow. As a proof-of-concept, the obfuscation techniques are implemented in BAMBU [19], an open-source HLS framework and applied the resulting flow to benchmarks that are much larger than the ones commonly used for gate-level obfuscation.

1.1 Contributions

Complete HLS solutions for obfuscating an IC are unavailable. However, this is a promising approach to target complex designs, while protecting the semantics of an IC more efficiently. The obfuscation techniques discussed in this chapter are imposing constraints on both the compiler front-end and the HLS engine to expose the elements to obfuscate instead of spreading them inside the design. Working at the HLS level allows to remove the sensitive algorithmic information and combine it with the locking key. The main contributions of the proposed approach are:

- The attack scenario consists in the untrusted foundry as the adversary in an oracle-less threat model for low-volume customers (Section 4);
- The approach embraces a set of obfuscation techniques that address the different elements of an algorithm to protect (Section 5.1);
- The techniques are integrated in a HLS-based design flow for algorithmic obfuscation that starts directly from C code (Section 5.2);
- Different solutions to manage the locking key are presented (Section 5.3).

The approach is evaluated by applying it to common HLS benchmarks, showing promising results for high-level obfuscation similar to program obfuscation.

1.2 Roadmap

After introducing logic locking (Section 2) and presenting the model of the components that we aim at protecting (Section 3), we introduce our threat model (Section 4. Then, we present our approach for algorithm-level obfuscation, showing how it is implemented in a HLS flow (Section 5). In Section 6, we evaluate the area and performance overhead for the obfuscation techniques and present a validation of the obfuscated designs.

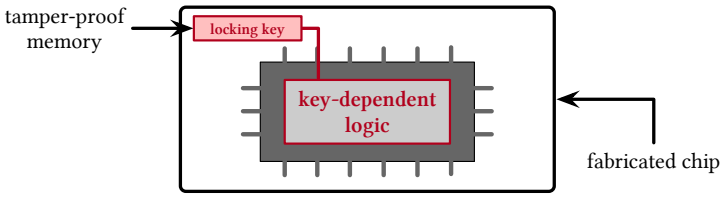

Fig. 2 Logic locking requires an additional input to deliver the *locking key* to the IC.

2 Background on Logic Locking

IC counterfeiting is a critical issue for *fabless* companies since they may lose billions of dollars for IP theft and overselling [9]. Addressing the problem, several IP protection techniques have been proposed at different stages of the design process [24]. To protect the intellectual property of an IC, the designer has to leverage intrinsic hardware properties of the device with Physical Unclonable Functions (PUFs) [16] or to modify the manufacturing process to separate the fabrication of the interconnections from the rest of the chip [12]. However, the process requires a 2.5D integration technology. Furthermore, these solutions require an intimate knowledge of the target technology and the back-end tool-chain.

Different solutions have been proposed for adding a "signature" to verify the ownership of an IC during litigation (*watermarking*) [3, 21]. Most of these approaches operate on the gate-level netlist or during layout generation [5, 13, 14], aiming at embedding a unique "signature". Also in this case, approaches have been proposed to raise the abstraction level for IP watermarking and operate during the component generation [1, 18, 26]. However, these methods are passive and cannot be used to prevent an illegal IC copy.

Logic locking is a well-known technique to thwart a potential attacker that wants to reverse engineer and copy the IC design. To do so, it hides the IC functionality against reverse engineering by using extra logic controlled by a key known only to the designers [25, 29, 28]. High-level transformations have been already proposed but only to obfuscate DSP circuits [15]. To thwart attacks aiming at recovering the key, several methods have been proposed to improve logic locking at the gate level [30, 33] or to raise the abstraction level and perform obfuscation during HLS [34], aiming at removing semantic information like in *program obfuscation* [32].

SAT-based attacks can extract these keys [31, 28]. In [22], the authors propose RTL hardening techniques by adding extra connections among the functional units. While this approach is more powerful than gate-level methods, constant values and branches are challenging to obfuscate since the design is already optimized. For instance, interconnections between resources and multiplexers have been sized based on the given precision. However, this reveals information on their range. Since we operate at a higher level of abstraction, it masks sensitive details of the algorithm by hiding sensitive constants and encrypting them during the front-end with a limited

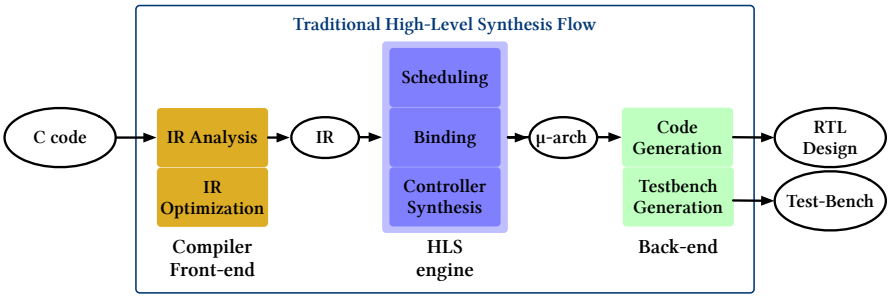

Fig. 3 Classic HLS flow.

overhead. Key management is another aspect of algorithmic obfuscation. Many companies are proposing solutions to store keys in tamper-proof memories (e.g., one-time-programmable memories) off-chip. These approaches are complementing the approach we present here wherein the keys are stored in on-chip tamper-proof and non-volatile memories.

3 Design Model and High-Level Synthesis

The approach we present generates RTL components with logic locking using HLS. The accelerator model couples a *controller* and a *datapath*, as in the *Finite-State Machine with Data path* (FSMD) model [6]. The *controller* is a finite state machine (FSM) that determines which operations execute in each clock cycle based on the evaluation of certain data-dependent conditions. Based on the set of operations to execute in each given clock cycle, the controller sends the proper control signals to trigger the functional units, the registers, and the interconnections in the *datapath* to drive the data values and perform the computation. Both parts are required to replicate the IC's functionality.

High-level synthesis (HLS) is a design methodology that allows designers to automatically derive an RTL design from its high-level description. The classic HLS flow is shown in Figure 3, assuming the input functionality is specified in C language. The HLS tool leverages state-of-the-art compilers (e.g., GCC or LLVM) to parse the input code, apply compiler optimizations (like loop transformations), and extract a language-independent intermediate representation (IR) [17]. The core HLS steps manipulate this IR as follows. *Scheduling* determines the operations to execute in each clock cycle based on data dependencies and the available hardware resources (e.g., functional units and memories). During *module binding*, operations scheduled in different clock cycles are analyzed for potential resource sharing to reduce area occupation. Data values crossing the clock boundaries are assigned to registers (*register binding*) [27]. In *interconnection binding*, the different resources (functional units, registers and memories) are interconnected and multiplexers are

added to correctly drive the signals when multiple data sources share the same target port. Ultimately, all resources are analyzed to derive the control signals needed in each clock cycles and the FSM controller is accordingly generated during *controller synthesis*. The output is an RTL design in Verilog or VHDL ready for logic synthesis.

4 Threat Model: The Untrusted Foundry

4.1 Untrusted Foundry's Objective

The main goal of the rogue in the untrusted foundry is to reverse engineer and replicate the target IC. For doing this, adversaries aim at recovering the correct sequence of states executed by the controller (execution traces) corresponding to given input sequences, along with the corresponding signals provided to the datapath (operations to execute, registers, and interconnections) in each given clock cycle. Once the entire design is recovered, the foundry can reproduce the component, thus misappropriating the IP. In case of designs protected with logic locking, this reverse-engineering process requires to identify also the correct locking key to obtain a working IC copy. So, the attacker aims at determining the design alternatives based on the values of the key bits, eventually ruling out implausible key values, i.e., values for which the design becomes clearly wrong.

4.2 Foundry's Capabilities

The semiconductor design houses use logic synthesis and physical design tools on the HLS results to obtain the GDSII file (i.e., the layout) of the IC to fabricate. The layout is then provided to the untrusted foundry for fabrication. However, the rogue in the untrusted foundry can access the GDSII file also to reverse engineer the functionality, attempting to break logic locking. To do so, we assume that the foundry can reverse engineer the types of modules used in the design (i.e., registers, functional units, interconnection elements) and can identify the operations executed by each functional unit (i.e., arithmetic, relational, and logic operations). The foundry can also perform simulations with different input and locking key values to extract information from the circuit that can help rule out implausible key values. However, the untrusted foundry does *not* have access to the correct key or a functioning unlocked IC (*oracle-less* attacks).

4.3 Target of the Attacks

The oracle-less attacks considered in this chapter are common in low-volume customers who build sensitive designs (e.g. US DoD). These designs are typical targets for attacks from untrusted foundries under pressure from their government to acquire proprietary cutting-edge technology. Until recently, IBM was maintaining the trusted foundry for the US government. Once it got acquired by Global Foundries (owned by a entity outside US), there is no trusted US foundry anymore, demanding effective methods for IC protection.

5 High-Level Synthesis Techniques for Algorithm Obfuscation

To protect the semantic and, in turn, the intellectual property (IP) of an algorithm via obfuscation, it is necessary to protect the following elements:

- *constant values* contains proprietary information (e.g., coefficients) or reveals details of the algorithm (e.g., loop bounds).
- *data-flow* describes how many and which operations are executed in each clock cycle together with their dependencies (i.e., which values are elaborated. This information is represented by the scheduled Data-Flow Graph (DFG).
- *control-flow* represents the sequence of FSM states traversed during the execution for the given inputs. It represents protocol implementations in control-dominated applications.

The elements must be obfuscated also to prevent further logic-level optimizations that can reveal proprietary information. For example, constant values are propagated to simplify the logic. Also, all elements must be obfuscated because they are connected and leaking information on one set of elements can aid recover details on the others. For example, multiplications by a constant that is power of two is often converted into shift operations that are more hardware-friendly. The optimization results can leak information both on the original operation (i.e., the multiplication) and the constant value (i.e., the power-of-two value).

The obfuscation techniques that we consider follow the same principles as in *program obfuscation* [4]. The real functionality is hidden with the creation of *opaque variables* or *opaque predicates*. A variable is opaque if it has some property (e.g., a value) that is known during obfuscation but is difficult for the attacker to deduce. Similarly, a predicate is opaque if its outcome is known only during obfuscation. To create opaque variables and predicates, expressions are combined with the locking key bits values that are known during obfuscation, but unknown to the attacker.

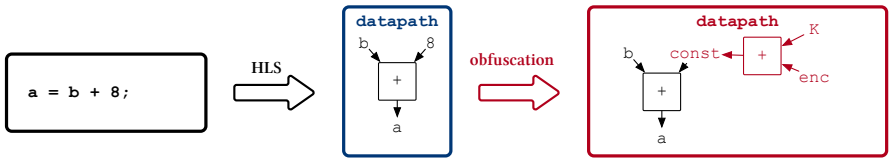

Fig. 4 Design modifications to implement constant obfuscation during RTL generation.

5.1 Obfuscation Techniques

In this section, the techniques proposed to obfuscate the elements discussed above is presented. All these elements require specific bits for obfuscation.

5.1.1 Constant obfuscation

Constant values are an essential part of the IP specification and may disclose sensitive information about the implemented algorithm. Consider a component with proprietary coefficients that realizes a specific digital filter. Such coefficients include also the number of taps in the filter. Without having this information, it is possible to replicate the type of component, but it is almost impossible to match the exact same filter function. Also, HLS tools optimize the datapath based on the data bit-width to reduce the IC cost [8, 17]. However, using the minimum number of bits to represent a constant leaks information about its range.

To enable constant obfuscation, sensitive constants in the datapath are replaced with opaque variables as shown in Fig. 4. The variable associated with a constant V_i is obtained by combining a value encoded in the circuit V_i^e with the locking key bits K_i as:

$$V_i^p = V_i^e \oplus K_i \tag{1}$$

where V_i^e is the obfuscated value that will be stored in the RTL micro-architecture, while K_i is a C-bit signal that represents the part of the working key dedicated to obfuscating the constant V_i. The encoded value associated with each constant V_i of the input algorithm is obfuscated as

$$V_i^e = V_i \oplus K_i \tag{2}$$

Clearly, the correct value is re-obtained only when the correct key is provided. Instead, if a wrong key is provided, the resulting value will be different from the one contained in the initial specification, but an attacker cannot determine this. Even when the constant represents a loop bound, the exact number of execution clock cycles for complex specifications is unknown to the attacker.

The number of bits C to implement all the constants of the function is pre-defined. The use of a pre-defined number of bits for all constants (regardless of their real size) hides the real bit-width of the specific constant, thwarting the identification

ALGORITHM 1: Algorithm to create DFG variants.

```
Procedure CreateDFGvariant(DFG_i, k_i)
    Data: DFG_i is the DFG of the basic block BB_i; k_i represents the key bits assigned to BB_i
    Result: VDFG_i is the set of DFG variants associated with BB_i
    Variants ← ∅
    V ← ComputeKeyVariants(k_i)
    foreach v ∈ V do
        dist_v ← ComputeDistance(v, k_i)        // compute distance between v and k_i
        DFG*_i ← CopyDFG(DFG_i)                 // create a copy of the current DFG
        OP ← ClusterOperations(DFG*_i)
        foreach op ∈ OP do
            op_j ← GetOperation(op, dist_v)     // return an operation at distance dist_v
              mod clusters
            SwapOperationTypes(op, op_j)        // statistically swap the types of the two
              operations
        end
        foreach dep ∈ DFG*_i do
            dep_j ← GetDependence(d, dist_v)    // return a dependence at distance dist_v
            RearrangeDependence(dep, dep_j) // statistically reorganize the dependences
        end
        Variants ← Variants ∪ DFG*_i
    end
    return Allocation
```

of the correct range. However, extracting the constants from the circuit may rule out subsequent logic optimizations (e.g., constant propagation and logic trimming), increasing the obfuscation cost.

> **Example.** Consider a constant $V_i = 10$ to be stored using 5 bits (`5'b01010`). The same value can be obfuscated as an 8-bit value as $V_i^e =$ `5'b01010111` or $V_i^e =$ `5'b11001101` based on locking keys $K_i =$ `5'b01011101` and $K_i =$ `5'b00100111`, respectively. □

This example also shows that the same constant V_i is encoded in different ways (resulting in different V_i^e values) based on the specific value of the locking key. This prevents the attacker from breaking the obfuscation and recovering the constant by comparing different versions of the design.

This step can be applied at any steps of the design flow. However, it is applied at the beginning, right after the compiler phase, to avoid constant-related HLS transformations and optimizations.

5.1.2 Data-flow obfuscation

To hide the arithmetic operations performed in the datapath, several DFG variations for each basic block are created. The DFG variations are based on the results of the HLS engine. Indeed, each basic block is schedule to determine the minimal number of functional units and registers to perform the computation, along with the latency (i.e., number of clock cycles), to perform the corresponding computation. Algorithm 1 shows how this information is used as constraints for creating the set $Variants$ of DFG variations starting from a valid schedule DFG_i and the k_i key bits assigned to the basic block b_i. The number of key bits assigned to the basic block b_i is proportional to the number of operations in b_i. In this way, a large number of variants is created only for more data-intensive basic blocks, while small

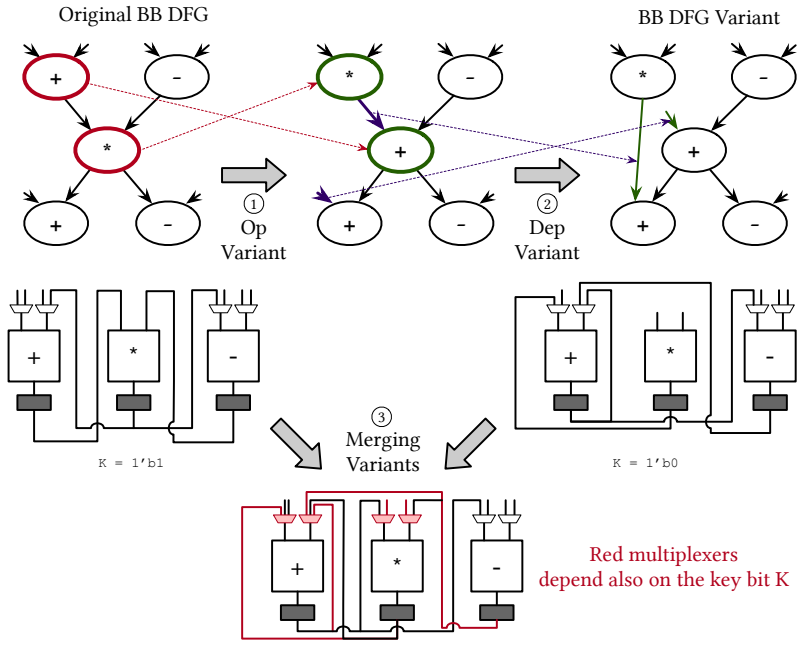

Fig. 5 Generation of DFG variants for operation obfuscation.

basic blocks (with less sensitive operations) are left untouched. First, the 2^{B_i-1} key variants are computed, beginning from the given key bits k_i. These key values will be associated with each DFG variant (`ComputeKeyVariants`) and values have the same number of bits as k_i but different values to distinguish the variants from the correct functionality. Then, for each variant, the Hamming distance between the corresponding key value and the obfuscation key bits k_i (`ComputeDistance`) is computed. A copy of the current schedule (`CopyDFG`) is produced, the operations are topologically ordered and then clustered based on the operation types. For each operation, the distance values are used as parameters for selecting the operations to alter and determining a reciprocal one in an alternative cluster `GetOperation`). Once the operations are selected, the two types are swapped with probability 0.5 (`SwapOperationTypes`). The algorithm proceeds to change DFG dependences in a similar way (`RearrangeDependences`). The set of resulting DFGs are merged together to create a single datapath micro-architecture, where multiplexers are inserted to drive the signals and implement one of the variants based on the values of the key bits.

Fig. 5 shows the application of this algorithm to a simple example. Starting from a DFG, pairs of different operations are selected for swapping their operand (step ① in Fig. 5). For every DFG edge, an alternative edge is selected and the dependencies to return a credible DFG (step ② in Fig. 5) are restructured. Finally, the architectures corresponding to each variant are recombined into a single data path microarchitecture, restructuring the interconnections using extra multiplexers

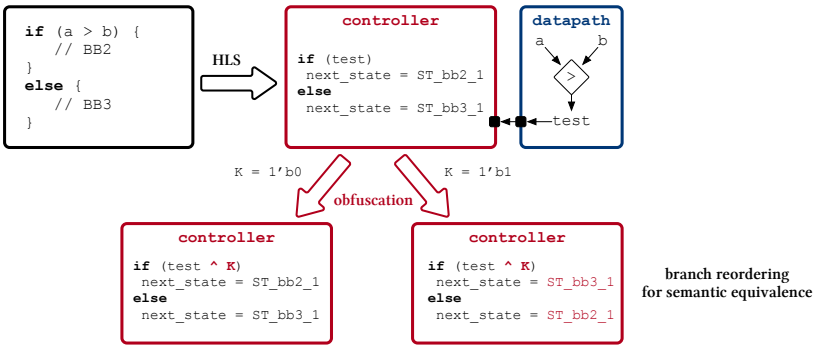

Fig. 6 Obfuscation of control branches. Different versions are obtained by combining the test with the assigned key bit, thwarting identification of the correct `true` and `false` blocks.

and control signals (step ③ in Fig. 5). In each clock cycle, the functionality to execute is selected through a combination of key bits (to select the variant) and scheduling information (to select the operations).

5.1.3 Control-flow obfuscation

Each branch in the CFG corresponds to a branch also in the corresponding controller FSM to determine the next state to execute. The target state depends upon the outcome (either true or false) of a predicate evaluation. The predicate is computed in the datapath (e.g., an arithmetic comparison or a Boolean operation) but is evaluated in next-state function of the controller. The identification of the correct condition (i.e., `true` and `false`) is thwarted and, in turn, the corresponding target state is also thwarted by assigning a key bit K_j to each branch j and changing the corresponding test in the controller to be of the form:

$$\texttt{test} \oplus K_j == 1'b1 \tag{3}$$

To maintain the semantic equivalence of the branch, the two branches are reordered based on the value of the key bit K_j. For instance, the `true` and `false` blocks are swapped when $K_j = 1$ because the `xor` operation inverts the value of the variable `test`. This transformation corresponds to the creation of an opaque predicate because the result of the `xor` is known during obfuscation because it is known the value of the key bit. On the contrary, the attacker cannot determine which is the actual `true` (`false`) block without knowing the value of the key bit. Fig. 6 shows this transformation on a simple example.

> **Example.** Consider the `if-then` statement in the black box shown in Fig. 6. When `a` is greater `b`, the control transfers to `BB2`, otherwise it transfers to `BB3`. After performing traditional HLS, we obtain the controller and data path shown in the red boxes of Fig. 6. Based on the results of the test, the next state is the first state of `BB2` or `BB3`. An attacker can determine which part of the algorithm executes when the condition is `true`. Conversely, our obfuscation technique can yield alternative versions of the controller (shown in the blue boxes in Fig. 6). The two resulting tests are perfectly equivalent, but

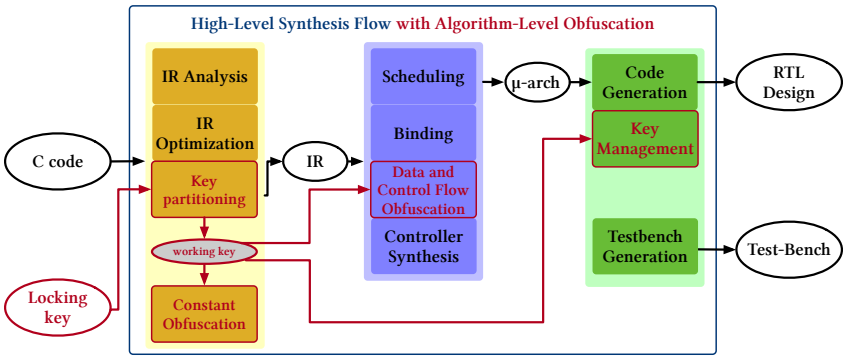

Fig. 7 HLS flow extended with key-based obfuscation.

the target state in case of `true` (`false`) result is different based on the key bit. So, the attacker cannot determine which is the real `true` block without knowing the correct value of the key bit. □

The same transformation applies to the test conditions of the `for`/`while` loops because the front-end compiler translates them into an identical form. One can obfuscate also complex branch constructs (e.g,. `switch`) by using more key bits.

5.2 Obfuscation Approach

Since the components generated with HLS require a strict interaction between datapath and the controller, an implementation of the approach should necessary be comprehensive, embracing all HLS steps to automatically implement the obfuscation transformations presented in Section 5. Such a comprehensive solution has been implemented extending the traditional HLS flow (see Section 3), making reverse engineering and hence the IP theft more difficult. This enhanced flow is shown in Fig. 7 and starts from the C code of the component to generate and the *locking key* K, which is generated by the designer to activate the IC after fabrication. The final output is a locked RTL design (with an extra input for providing the key) ready for logic synthesis and physical design.

5.2.1 Compiler Front-end

The input C code is processed with the compiler front-end to general the internal intermediate representation (IR). Then, common compiler transformations, including function inlining and loop optimizations, are applied to the IR to prepare it for HLS. Constant propagation is instead disabled so that the information can be obfuscated.

The IR generated and optimized during the HLS front-end is processed to determine the number of key bits needed to obfuscate the different elements of the algorithm. For this, the call graph is extracted to figure out the list and hierarchy

of functions to synthesize [17]. Other information consists of the number of basic blocks[1] and the resulting Control Flow Graph (CFG). A fixed number C of key bits is assigned to obfuscate each constant, B_i key bits to obfuscate the scheduled DFG of each basic block b_i, and one bit to obfuscate each control branch. By combining this information with the IR details, it is determined the number of bits of the internal obfuscation key W, called *working key*, needed to obfuscate the algorithm:

$$W = Num_{if} + Num_{const} * C + \sum_{i=0}^{BB} B_i \tag{4}$$

where Num_{if} and Num_{const} are the number of branches and constants, respectively. C is the number of key bits assigned to implement each constant and B_i is the number of key bits assigned to the basic block BB_i. So, the size of the working key W depends on the complexity of the algorithm to protect and is usually larger than the locking key K provided by the designer. Section 5.3 describes how to generate the working key K starting from the input locking key K.

5.2.2 HLS Engine

In the mid-level phase, we perform the traditional HLS steps, extended with the obfuscation techniques. First, the constants are extracted and obfuscated (see Section 5.1.1) to prevent HLS transformations and optimizations based on their bit-widths and values. For example, multiplications by constants are usually simplified to obtain more efficient hardware [2]. However, this may reveal sensitive information that cannot remove after HLS. The resulting IR is input to the HLS steps to create the datapath and the controller of each sub-function.

For creating the datapath, after scheduling each basic block, several variants are created with the goal of thwarting the identification of the arithmetic operations and dependencies (see Section 5.1.2). Since B_i obfuscation bits are used for the basic block b_i, the corresponding key value is assigned to the correct version and the other $2^{B_i} - 1$ values to the variants. In the resulting datapath microarchitecture, extra connections between functional units and registers are added to implement the functionality of the different variants, and multiplexers to activate the execution of the variant associated with the value of the corresponding key portion.

For creating the controller, the FSM associated with each scheduled module is determined and each control branch is obfuscated (see Section 5.1.3). In case of a conditional jump, the result of the condition evaluation performed in the datapath is masked with a key bit. The next-state functions of the controller are thus masked with key bits to obfuscate the correct transitions while maintaining logical but incorrect execution flows in case of wrong locking keys.

[1] a basic block is a sequence of instructions with a single entry point and a single exit point.

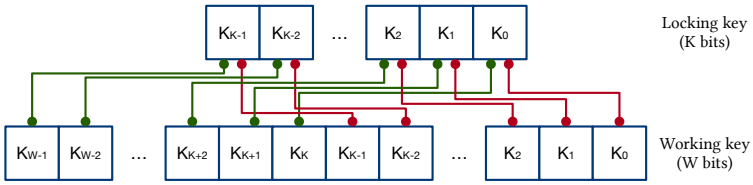

Fig. 8 Key management when working key is larger than the locking key ($W > K$). the locking key is repeated to create the working key.

At the end of the HLS steps, the module of each function is created by combining the corresponding datapath and controller. The hierarchy of the modules as in the traditional HLS flow is also created.

5.2.3 Back-end

This step generates the RTL description and the logic for key management of the obfuscated design. The component will feature an additional input port to load the locking key from the system, while the working key used for obfuscation is stored internally and derived from the input locking key. The strategy to manage locking and working keys are described in Section 5.3.

5.3 Key Management

5.3.1 Storing the locking key

The *locking key* is the only extra input used to lock the circuit, as shown in Fig. 2. This key is given to our obfuscation approach for applying the obfuscation techniques but not to the foundry. Instead, it is stored in a tamper-proof memory (e.g., EEPROM, eFuses or Non-Volatile Memory [7]) after IC fabrication [25, 22]. The number of *locking key* bits that one can deliver to the IC may be fixed and limited by the technology. On the contrary, the number of key bits needed by an algorithm for obfuscation (*working key*) depends on the number and size of the basic blocks, number of control branches, and number of constants) and the obfuscation techniques that are applied.

5.3.2 Generating the working key

When the number of working key bits is smaller than the number of available locking key bits, there is a one-to-one correspondence between the working and locking key bits, which are thus directly connected. This situation is ideal because each key bit is unique and there is no additional overhead for key management. However, this is not

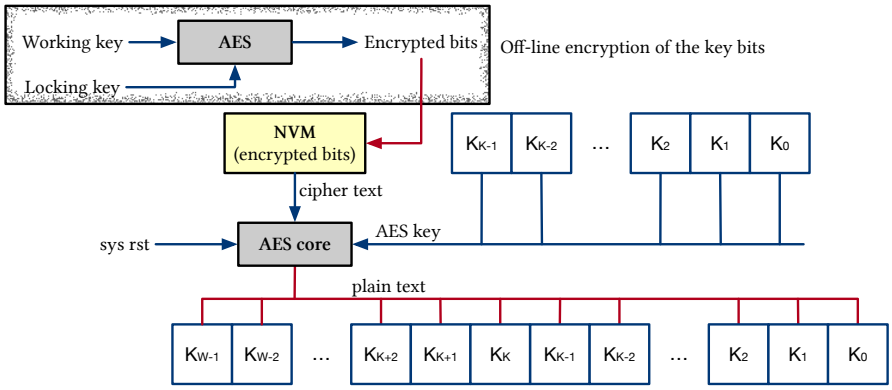

Fig. 9 Key management when working key is much larger than the locking key ($W >> K$). The key is generated at power up and AES decrypts the values in the non-volatile memory.

always possible, and cases where exist more working key bits than available locking key bits needs to be considered. When it is needed to derive many working key bits from a smaller number of locking key bits, one solution entails reusing the locking key bits as many times as needed to generate the working key, as shown in Fig. 8. In this situation, each key bit has a maximum fan-out of $f = \lceil W/K \rceil$, which may leak information to break the logic locking. Indeed, the attacker can use correlation analysis to extract information about each single key bit. If the attacker can extract the value of one working key bit, the corresponding locking key bit and, in turn, all its replicas are extracted. This may compromise the security of the generated IC for large values of f, i.e., when the number of working key bits is much larger than the number of available key bits.

For this case, an alternative solution is shown in Fig. 9. The locking key is used by an AES cryptographic core to encrypt the working key at design time. The resulting cipher-text is stored in a Non-Volatile Memory (NVM) that is added to the IC. At power-up, the NVM values are decrypted using the given locking key and placed into the working-key registers. So, the correct working key is loaded only when the correct locking key is placed into the IC and used for decryption. This solution does not introduce any performance overhead for deriving the working key since this process is performed only at power-up, when the system is not operational. Also, since this solution leverages the security guarantees of AES, we can use a 256-bit locking key, which is a reasonable size for existing technologies, to secure a large number of working key bits.

6 Experimental Evaluation

The obfuscation approach is validated by extending Bambu, an open-source HLS framework [19]. Since Bambu has a modular organization, the obfuscation tech-

niques is implemented as additional steps in the HLS flow. The resulting version of the tool is called LockBambu.

6.1 Experimental Setup

LockBambu is used to generate obfuscated circuits on five HLS benchmarks from a range of application domains: GSM is a linear predictive coding analysis for telecommunication. ADPCM is an algorithm for adaptive differential pulse code modulation, SOBEL is an image-processing algorithm. BACKPROP is a method for training neural networks, and VITERBI is a dynamic programming method for computing probabilities on a Hidden Markov model. These algorithms represent applications that a designer may want to obfuscate because of proprietary implementations. Table 1 shows the characteristics of the benchmarks.

Table 1 Characteristics of the benchmarks.

Benchmark	# C lines	# Const	# BB	# CJMP	W (bits)
GSM	110	4	88	4	484
ADPCM	412	5	100	5	565
SOBEL	65	2	11	2	110
BACKPROP	264	12	123	11	887
VITERBI	144	117	98	9	4,145

For each benchmark, Table 1 reports the number of constants (`# Const`), basic blocks (`# BB`), and control branches (`# CJMP`) following the compiler optimizations. Together with the number of lines of C code (`# C lines`), they capture the algorithm complexity. The bit-width of each obfuscated constant is set to 32 bits (i.e., $C = 32$), while the original constants range between 8 (`char` values) and 32 bits (`int` values). One bit is assigned to each control branch. Finally, four bits are assigned to each basic block to generate up to 16 DFG variants (i.e., $B_i = 4$ for all basic blocks) Table 1 reports the working key bits required for each algorithm (`W`). The resulting number of working key bits shows that constant obfuscation requires a large number of bits. For example, VITERBI and GSM have more or less the same number of basic blocks and control branches but the former has many more constants, requiring 10× more key bits than the latter. These benchmarks are bigger than those commonly used for logic obfuscation. Working at a higher abstraction allows us to obfuscate larger circuits.

For each benchmark, 256-bit locking keys are generated and the effects of the obfuscation technique are evaluated in terms of *obfuscation potency* (how much is the attacker confused?) and *obfuscation cost* (what is the obfuscation overhead?). To evaluate the obfuscation techniques, the baseline designs (generated with Bambu) are compared with the corresponding obfuscated ones (generated with LockBambu).

Bambu generates RTL testbenches to validate the circuit for a series of input values through RTL simulations. These executions are compared against the respective executions of the input specification in software. In LockBambu, these testbenches are extended to specify different locking keys as input to verify the execution for each of them. Simulations are performed with Mentor ModelSim SE 10.3 and are instrumented to report if the execution provides the same results as the baseline design (to evaluate *obfuscation potency*) and the number of cycles (to evaluate to evaluate *obfuscation cost* in terms of performance overhead). The baseline and obfuscated versions of the circuits are synthesized using Synopsys Design Compiler J-2014.09-SP2 targeting the Synopsys SAED 32nm Generic Library at 500 MHz (to evaluate *obfuscation cost* in terms of area overhead).

6.2 Evaluation of Obfuscation Potency

For each benchmark, 100 random keys are generated. The first key is used as input for LockBambu (locking key), while the others are used to test for security evaluation. First, the generated circuits are simulated with the correct locking key corroborating that the circuits produce the same results as in the baseline version. All other keys result in wrong results and this assures that the attacker cannot turn on the circuit with another key. The obfuscation potency (i.e., how much attacker is confused) is quantified using the "output corruptibility" of each locked circuit, computed as the Hamming distance with respect to the output of the baseline circuit [31]. The ideal obfuscation procedure should provide an output corruptibility of 50% to avoid any bias in the output bits that can leak information on the key values. When combined, the three obfuscation techniques produce an average HD of 62.2% over the five benchmarks, which is a good result. Also, designs can leak information through variations in the execution latency (*timing channels*). However, incorrect locking keys impact the performance only when they modify the loop bounds. Other constants have no effect, while data path obfuscation works on a valid schedule without altering the total number of cycles. It is difficult for an attacker to tell whether a circuit is behaving properly or not. While the alternative DFGs are conceptually similar to the creation of the Super CDFG [22], SAT-based attacks are much harder to apply because the oracle chip is unavailable in the untrusted foundry threat model and the complexity of the circuits demands novel methods to apply these attacks on large sequential circuits composed of datapath and controller. Moreover, in case of constants, the information is fully cut out from the circuit, and one cannot recover it without the correct locking key. In conclusion, the circuits generated by LockBambu have a higher security level than previous obfuscation techniques operating at the logic level.

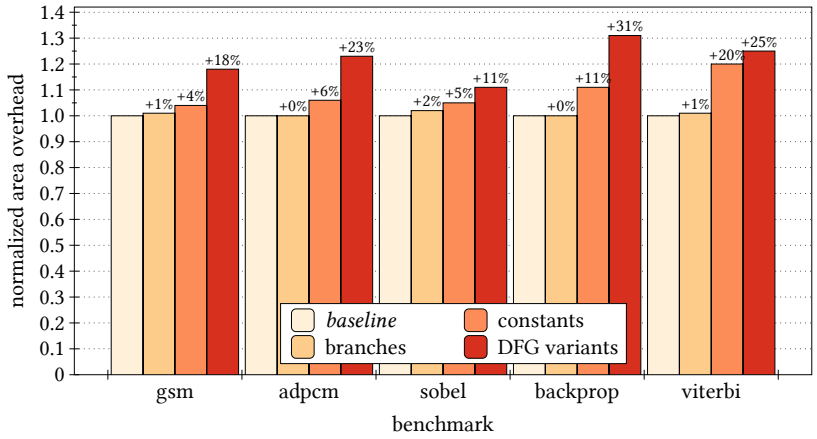

Fig. 10 Area overhead of obfuscation techniques.

6.3 Evaluation of Obfuscation Cost

To evaluate the cost of each obfuscation technique, LockBambu is modified to selectively apply the methods through command-line options. Since these transformations are orthogonal, different versions of the circuits are generated for each of them.

RTL simulations are performed to check the *performance overhead* concerning the circuit latency (in clock cycles). When the correct key is applied, there is no performance overhead on the generated designs concerning the *baseline* versions. However, the target frequency is decreased by 8% on average when we apply data-flow obfuscation because of the more additional multiplexers. Also, the drop off in frequency is proportional to the number of key bits assigned to each basic block because creating more variants requires more multiplexers. Obfuscating the control branches has a negligible impact on the frequency (less than 1%). Representing the constants by a pre-defined number of bits C increases the size of multiplexers and reduces logic optimizations. However, the impact the critical path is minimal (around 4%). When all obfuscations are applied, the target frequency is decreased by less than 10% on average that can be reduced to 6% on average with more aggressive logic synthesis optimizations.

Logic synthesis is carried out to evaluate the area overhead of the various obfuscation techniques. Fig. 10 shows the results, where each value is normalized against the area of the respective *baseline* version (obtained with the original version of Bambu). The results indicate that obfuscating the control flow has practically no area impact. This technique only adds a few exclusive-or gates to the controller. Obfuscating constants increases the area by 10% on average since it creates larger multiplexers and prevents logic-level optimizations. Data-flow obfuscation has the most impact, increasing the area by around 21% on average. This area overhead is mainly due to the additional multiplexers to connect functional units and registers. This obfuscation is appropriate for benchmarks where the computational part has

simple functional units (e.g., shifters and Boolean operations) or has many basic blocks. BACKPROP is the benchmark with more basic blocks and has the largest overhead (>30%). Similarly to the frequency, the area overhead is proportional to the number of key bits assigned to the basic blocks. When all obfuscation techniques are applied together, the overhead adds up, resulting in a total overhead between 20% and 45%. It is worth noting that memory controllers to access the external memory are responsible for a significant portion of the circuit area, but they are not obfuscated because their implementation is generic does not contain any algorithm-dependent part. This significantly reduces the impact of obfuscation on the final design.

In the basic approach of replicating the key bits, there is no performance or area overhead. The signals are coming from the tamper-proof memory where the locking key is stored and directly connects to the points where one uses the working key. For the AES-based solution, there are two contributions to the area overhead: one part is the AES decryption module, and the other one is the NVM used to store the encrypted key bits and the flip-flops to save the decrypted values. The first contribution is fixed and depends on the AES implementation. The second contribution is proportional to the number of working key bits. The key decryption is performed only once at power-up and there is no performance overhead once the chip is ready to use.

7 Conclusion

In this chapter, we present approach for implementing obfuscation during high-level synthesis that is able to hide the algorithm semantics to the attacker and thwart reverse engineering of the corresponding physical design. The presented approach starts from a C-level description of the algorithm and creates a version of the corresponding RTL component by masking all relevant algorithm portions through opaque predicates that are based on an input locking key. In particular, techniques for obfuscating constant values, arithmetic operations, and control branches have been presented. These techniques have been implemented within Bambu, a state-of-the-art, open-source HLS tool. This combination allowed us to obtain a comprehensive solution for obfuscation during HLS that has been validated on a set of representative benchmarks. These techniques do not incur performance overhead and each of them has a maximum area overhead of around 30% (20% on average).

Acknowledgments

R. Karri is supported in part by NSF (A#: 1526405) and CCS-AD. S. Garg is supported in part by an NSF CAREER Award (A#: 1553419). S. Garg and R. Karri are both with the NYU Center for Cybersecurity (cyber.nyu.edu) and supported in part by Boeing Corp.

References

1. Abdel-Hamid, A.T., Tahar, S., Aboulhamid, E.M.: IP watermarking techniques: Survey and comparison. In: Proceedings of the IEEE International Workshop on System-on-Chip for Real-Time Applications (IWSOC), pp. 60–65 (2003)
2. Boullis, N., Tisserand, A.: Some optimizations of hardware multiplication by constant matrices. IEEE Transactions on Computers **54**(10), 1271–1282 (2005)
3. Charbon, E.: Hierarchical watermarking in IC design. In: Proceedings of the IEEE Custom Integrated Circuits Conference (CICC), pp. 295–298 (1998)
4. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Tech. Rep. 148, Department of Computer Science, The University of Auckland, New Zealand (1997)
5. Cui, A., Chang, C.H., Tahar, S., Abdel-Hamidothers, A.T.: A robust FSM watermarking scheme for IP protection of sequential circuit design. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **30**(5), 678–690 (2011)
6. De Micheli, G.: Synthesis and Optimization of Digital Circuits. McGraw-Hill (1994)
7. Forte, D., Bhunia, D., Tehranipoor, M.: Hardware Protection Through Obfuscation. Springer Publishing Company, Incorporated (2017)
8. Gal, B.L., Andriamisaina, C., Casseau, E.: Bit-width aware high-level synthesis for digital signal processing systems. In: Proceedings of the IEEE International SOC Conference (SOCC), pp. 175–178 (2006)
9. Guin, U., Huang, K., DiMase, D., Carulli, J.M., Tehranipoor, M., Makris, Y.: Counterfeit Integrated Circuits: A rising threat in the global semiconductor supply chain. Proceedings of the IEEE **102**(8), 1207–1228 (2014)
10. Heck, S., Kaza, S., Pinner, D.: Creating value in the semiconductor industry. McKinsey on Semiconductors pp. 5–144 (2011)
11. Hurtarte, J., Wolsheimer, E., Tafoya, L.: Understanding Fabless IC Technology. Elsevier (2007)
12. Imeson, F., Emtenan, A., Garg, S., Tripunitara, M.: Securing computer hardware using 3D integrated circuit (IC) technology and split manufacturing for obfuscation. In: Proceedings of the USENIX Conference on Security (SEC), pp. 495–510 (2013)
13. Kahng, A.B., Lach, J., Mangione-Smith, W., Mantik, S., Markov, I.L., Potkonjak, M., Tucker, P., Wang, H., Wolfe, G.: Constraint-based watermarking techniques for design IP protection. IEEE Trans. Comput. Aid. Des. **20**(10), 1236–1252 (2001). DOI 10.1109/43.952740
14. Kahng, A.B., Mantik, S., Markov, I.L., Potkonjak, M., Tucker, P., Wang, H., Wolfe, G.: Robust IP watermarking methodologies for physical design. pp. 782–787. ACM (1998)
15. Lao, Y., Parhi, K.K.: Obfuscating DSP circuits via high-level transformations. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **23**(5), 819–830 (2015)
16. der Leest, V.V., Tuyls, P.: Anti-counterfeiting with hardware intrinsic security. In: Proceedings of the Design, Automation & Test in Europe Conference (DATE), pp. 1137–1142 (2013)
17. Nane, R., Sima, V., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y.T., Hsiao, H., Brown, S., Ferrandi, F., Anderson, J., Bertels, K.: A Survey and Evaluation of FPGA High-Level Synthesis Tools. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **35**(10) (2016)
18. Pilato, C., Basu, K., Shayan, M., Regazzoni, F., Karri, R.: High-level synthesis of benevolent trojans. In: Proceedings of the Design, Automation & Test in Europe Conference (DATE), pp. 1124–1129 (2019)
19. Pilato, C., Ferrandi, F.: Bambu: A modular framework for the high level synthesis of memory-intensive applications. In: Proceedings of the International Conference on Field programmable Logic and Applications (FPL), pp. 1–4 (2013)
20. Pilato, C., Garg, S., Wu, K., Karri, R., Regazzoni, F.: Securing hardware accelerators: A new challenge for high-level synthesis **10**(3), 77–80 (2018)
21. Qu, G., Potkonjak, M.: Intellectual property protection in VLSI designs: theory and practice. Kluwer Academic (2003)
22. Rajendran, J., Ali, A., Sinanoglu, O., Karri, R.: Belling the CAD: Toward security-centric electronic system design. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **34**(11), 1756–1769 (2015)

23. Reuters Semiconductors: TSMC says latest chip plant will cost around $20 bln. Available at: https://www.reuters.com/article/tsmc-investment/tsmc-says-latest-chip-plant-will-cost-around-20-bln-idUSL3N1O737Z (2017)
24. Rostami, M., Koushanfar, F., Karri, R.: A primer on hardware security: Models, methods, and metrics. Proceedings of the IEEE **102**(8), 1283–1295 (2014)
25. Roy, J.A., Koushanfar, F., Markov, I.L.: Ending Piracy of Integrated Circuits. Computer **43**(10), 30–38 (2010)
26. Sengupta, A., Roy, D.: Antipiracy-aware ip chipset design for ce devices: A robust watermarking approach [hardware matters]. IEEE Consumer Electronics Magazine **6**(2), 118–124 (2017)
27. Stok, L.: Data path synthesis. Integration VLSI Journal **18**(1), 1–71 (1994)
28. Subramanyan, P., Ray, S., Malik, S.: Evaluating the security of logic encryption algorithms. In: Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pp. 137–143 (2015)
29. Vijayakumar, A., Patil, V.C., Holcomb, D.E., Paar, C., Kundu, S.: Physical design obfuscation of hardware: A comprehensive investigation of device and logic-level techniques. IEEE Transactions on Information Forensics and Security **12**(1), 64–77 (2017)
30. Xie, Y., Srivastava, A.: Anti-SAT: Mitigating SAT attack on logic locking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **38**(2), 199–207 (2019)
31. Xie, Y., Srivastava, A.: Anti-SAT: Mitigating SAT attack on logic locking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **38**(2), 199–207 (2019)
32. Xu, H., Zhou, Y., Kang, Y., Lyu, M.R.: On secure and usable program obfuscation: A survey. In: ArXiv (2017)
33. Yang, F., Tang, M., Sinanoglu, O.: Stripped Functionality Logic Locking with Hamming Distance Based Restore Unit (SFLL-hd) – unlocked. IEEE Transactions on Information Forensics and Security pp. 1–9 (2019)
34. Yasin, M., Zhao, C., Rajendran, J.J.: SFLL-HLS: Stripped-functionality logic locking meets high-level synthesis. In: Proceeding of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1–4 (2019)