TAO: Techniques for Algorithm-Level Obfuscation during High-Level Synthesis

Christian Pilato and Francesco Regazzoni
Università della Svizzera italiana, Lugano, Switzerland
christian.pilato@usi.ch,francesco.regazzoni@usi.ch

Ramesh Karri and Siddharth Garg
New York University, New York, NY, USA
rkarri@nyu.edu,sg175@nyu.edu

ABSTRACT

Intellectual Property (IP) theft costs semiconductor design companies billions of dollars every year. Unauthorized IP copies start from reverse engineering the given chip. Existing techniques to protect against IP theft aim to hide the IC's functionality, but focus on manipulating the HDL descriptions. We propose TAO as a comprehensive solution based on high-level synthesis to raise the abstraction level and apply algorithmic obfuscation automatically. TAO includes several transformations that make the component hard to reverse engineer during chip fabrication, while a key is later inserted to unlock the functionality. Finally, this is a promising approach to obfuscate large-scale designs despite the hardware overhead needed to implement the obfuscation.

CCS CONCEPTS

• **Hardware** → *Electronic design automation*; • **Security and privacy** → *Hardware reverse engineering*;

KEYWORDS

High-Level Synthesis, Algorithm-Level Obfuscation, Reverse Engineering, IP theft.

1 INTRODUCTION

The cost of manufacturing an Integrated Circuit (IC) is growing as technology scales, limiting the number of companies that can afford the billion dollar manufacturing foundries [6]. Many companies are thus becoming *fab-less*, outsourcing IC manufacturing to third-party foundries [7]. This creates security issues: a rogue in the potentially untrusted foundry can access the chip design and reverse engineer the functionality to steal the Intellectual Property (IP) [5].

Several techniques have been proposed to thwart reverse engineering of an IC at an untrusted foundry. Split manufacturing splits the computing resources from the interconnections, with the two parts fabricated in different foundries. *Logic obfuscation* has been extensively investigated for this purpose as well [14]. The designer adds additional inputs and modules to the design to hide the correct functionality, while a *locking key* (unknown to the foundry and written later in a tamper-proof memory) activates the IC. With the increasing complexity of ICs, designers are migrating to *high-level synthesis* (HLS) to automate the design process [11]. While one can apply logic obfuscation on the generated netlist [13], more robust solutions can be applied directly at the algorithm level.

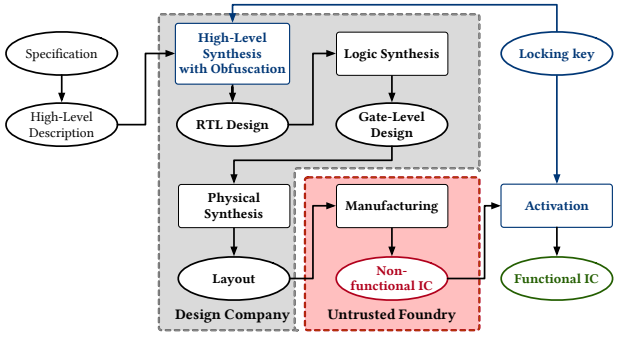

Figure 1: IC design flow; red artifacts show IC reverse engineering. TAO extends HLS with obfuscation using a key.

Algorithm-level obfuscation aims at developing anti-reverse engineering techniques based on the characteristics of the algorithm the different steps in HLS. We aim at raising the abstraction level of RTL obfuscation by embracing a security-aware HLS flow to generate obfuscated designs by construction.

Techniques for Algorithm Obfuscation (TAO) start 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. TAO extends HLS algorithms to obfuscate the most sensitive details of an algorithm. TAO presents techniques that obfuscate 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 data path resources, execution latency). TAO can obfuscate complex functions as part of a comprehensive HLS-based obfuscation design flow.

1.1 Related Work

IC counterfeiting is a critical issue for *fabless* companies since they may lose billions of dollars for IP theft and overselling [5]. So, several IP protection techniques have been proposed at different stages of the design process. Some methods focus on modifying the design before fabrication with hardware watermarking [1] or leveraging intrinsic hardware properties of the device with Physical Unclonable Functions (PUFs) [17]. However, these solutions require an intimate knowledge of the target technology and the back-end tool-chain. Split manufacturing separates the fabrication of the interconnections from the rest of the chip [8]. However, the process requires a 2.5D integration technology. Another solution hides the IC function by adding extra gates to the gate-level netlist and uses

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196126

a secret key to activate the IC [14],[10]. High-level transformations have been already proposed but only to obfuscate DSP circuits [9].

SAT-based attacks can extract these keys [16, 18]. In [13], the authors propose RTL hardening techniques by adding extra connections among the functional units. While this approach is more potent 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 TAO applies 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 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 this work wherein TAO stores the keys in on-chip tamper-proof non-volatile memories.

1.2 Contributions

HLS solutions to obfuscate an IC are unavailable. However, this is a promising approach to broaden the set of obfuscations and to obfuscate complex designs. For instance, in TAO, we propose obfuscations both at the front-end level and during HLS. Working at the HLS level allows us to remove the sensitive algorithmic information and integrate it within the key. The main contributions of TAO are as follows:

- TAO considers the untrusted foundry as the adversary (Section 3.1);
- TAO is a HLS-based design flow for algorithmic obfuscation that starts directly from C code (Section 3.2);
- TAO uses a set of obfuscation techniques that span all the HLS steps (Section 3.3);
- TAO shows how to manage the locking key (Section 3.4).

1.3 Roadmap

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

2 DESIGN MODEL

TAO targets ICs generated using HLS. State-of-the-art HLS tools rely on the *Finite-State Machine with Data path* (FSMD) model [2]. The *controller* is a finite state machine (FSM) that determines which operations execute in each clock cycle. The controller sends control signals to trigger the operators and the interconnection in the *data path* to perform the computation. These two parts are coupled, and both are required to extract IC's function.

The HLS flow is shown in Figure 2. It interfaces with state-of-the-art compilers (e.g., GCC or LLVM) to parse the input C code, apply compiler optimizations, and extract an intermediate representation (IR) [11]. The HLS steps work on this IR as follows. *Scheduling* selects the resources and memories and determines the operations

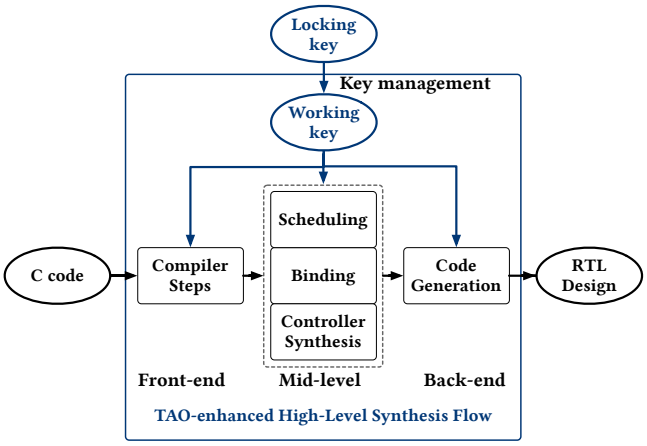

Figure 2: HLS flow extended with key-based obfuscations.

to execute in each clock cycle. During *module binding*, operations scheduled in different clock cycles are assigned to reuse resources. Temporary values crossing the clock boundaries are mapped to different registers during *register binding* [15]. The penultimate step in HLS is *interconnection binding*, where the different resources are interconnected. Ultimately, the control signals are identified and the controller is generated during *controller synthesis*. The output is an RTL design ready for logic synthesis.

To protect the intellectual property (IP) of an algorithm, we identify the following elements to protect via obfuscation:

- *arithmetic operations:* The designer aims at obfuscating the Data Flow Graph (DFG), i.e., which and how many operations are executed, together with their dependencies.
- *constant values (e.g., coefficients, loop bounds)* reveal details of the algorithm on one hand and enable further logic-level optimizations on the other. The optimization results can leak information on the operations manipulating these values.
- *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 are connected because leaking information on one set of elements can aid recover details on the others.

3 HIGH-LEVEL SYNTHESIS TECHNIQUES FOR ALGORITHM OBFUSCATION (TAO)

3.1 TAO Threat Model: The Untrusted Foundry

3.1.1 Untrusted Foundry's Objective. The main goal of the rogue in the untrusted foundry is to identify the functionality of the IC. In particular, he or she aims at recovering the correct sequence of states executed by the controller, along with the signals provided to the data path (operations to execute, registers, and interconnections) in each given clock cycle. This gives the foundry the possibility of reproducing the IC, thus misappropriating the IP.

3.1.2 Foundry's Capabilities. The untrusted foundry has full access to the GDSII file (i.e., the layout) of the obfuscated circuit generated from the synthesis output using physical design tools. From the GDSII file, 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. The foundry can also perform simulations with different input and lock key values to extract information from the circuit that can help reconstruct the functionality. However, the untrusted foundry does *not* have access to the correct key or a functioning unlocked IC.

3.1.3 Target of the Attacks. Low volume customers who build sensitive designs (e.g. US DOD) are typically targeted by untrusted foundries under pressure from their government. Until recently, IBM was maintaining the trusted foundry for the US government. Once it got acquired by Global Foundries owned by a foreign entity, there is no trusted foundry anymore.

3.2 TAO Approach

TAO extends the traditional HLS flow to obfuscate the IC functionality and make reverse engineering and hence the IP theft difficult. Since an HLS-generated component requires interaction between the data path and the controller, TAO is a comprehensive solution for algorithm obfuscation embracing all steps in HLS.

3.2.1 Front-end. A *locking key K* is generated by the designer to activate the IC, as shown in Fig. 1. The IR generated and optimized during the HLS front-end is processed to determine the *working key W* used for obfuscation. The size of the working key depends on the complexity of the algorithm to protect. We assign a fixed number of key bits to obfuscate each constant, each basic block (to obfuscate the DFG and the FSM states resulting from its scheduling), and each control branch. After compiler parsing and optimization steps, TAO extracts and obfuscates the constants (see Section 3.3.2) to prevent HLS optimizations based on their bit-width that may reveal sensitive information.

3.2.2 Mid-level. The IR is input to the HLS. The data path and controller of each sub-function are obfuscated to hide the execution of the correct algorithm. TAO obfuscates each control branch (see Section 3.3.3) and basic block (see Section 3.3.4). In case of a conditional jump, TAO masks the result of the condition with a key bit that obfuscates the target state. The output and next-state functions of the controller are masked with key bits to obfuscate the correct transitions while maintaining logical but incorrect execution flows in case of wrong locking keys. For each basic block, TAO creates several DFG variants to thwart identification of the arithmetic operations and dependencies. In the data path, we add extra connections among functional units and registers to implement several valid DFG variants. The choice of the variants is encoded by the key bits assigned to the basic block.

3.2.3 Back-end. This step generates the register transfer level (RTL) description and the logic for key management of the obfuscated design. The component will feature an input port to load the locking key, while the working key is stored internally and derived from the input locking key. We discuss how to manage the case where we need more working key bits than the available locking key bits (Section 3.4).

3.3 TAO Obfuscation Techniques

In this section, we present the techniques implemented in the front-end and the intermediate phases.

3.3.1 Creation of the Call Graph and Key Apportionment. TAO starts by applying compiler and HLS transformations to the IR, including function inlining and loop optimizations. For this, TAO extracts the call graph to figure out the list and hierarchy of functions implemented [11]. Other information consists of the number of basic blocks[1] and the resulting control flows represented as a Control Flow Graph (CFG). By analyzing this information, TAO determines the number of working key bits W needed to obfuscate the algorithm:

$$W = Num_{if} + Num_{const} * C + \sum_{i=0}^{BB} B_i \quad (1)$$

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.

3.3.2 Extraction of Constants. Constants are a requisite part of the specification and may disclose sensitive information about the implemented algorithm. Consider a digital filter whose coefficients are stored in an on-IC memory external to the component and accessed through a memory interface. The loop bounds may reveal the number of taps in the filter. TAO removes such sensitive constants from the data path and use them as locking key bits.

HLS tools optimize the data path based on the data bit-width to reduce the IC cost [4]. However, using the minimum number of bits to represent a constant divulges information about its range. TAO pre-defines the number of bits C to implement all constants of the function. This may rule out subsequent logic optimizations (e.g., constant propagation and trimming). Each constant V_i^p of the input algorithm is obfuscated as follows:

$$V_i^e = V_i^p \oplus K_i \quad (2)$$

where V_i^e is the obfuscated value stored in the micro-architecture, while K_i is a C-bit signal that represents the part of the working key dedicated to obfuscating this constant. As a result, the same constant value is coded in different ways based on the value of the locking key, preventing the attacker from recovering the sensitive information by comparing different versions of the design.

Example. Consider a constant $V_i^p = 10$ to be stored using 5 bits (`5'b01010`). The same value can be obfuscated as V_i^e = `5'b10111` or V_i^e = `5'b01101` based on locking keys K_i = `5'b11101` and K_i = `5'b00111`, respectively. The correct signal is obtained by combining the obfuscated values V_i^e with the input key bits:

$$V_i^p = V_i^e \oplus K_i \quad (3)$$

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.

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

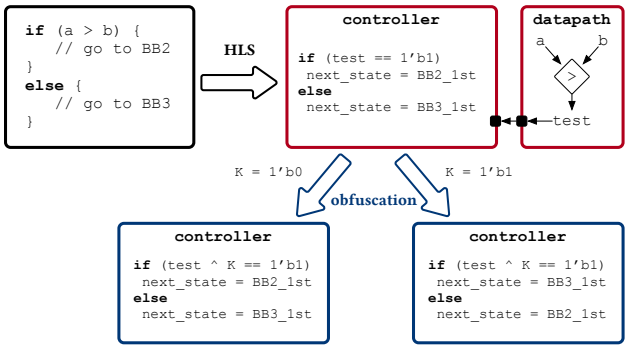

Figure 3: Obfuscation of control branches. Different versions are obtained by combining the test with the assigned key bit, thwarting identification of the correct `true` and `false` blocks.

3.3.3 Masking Control Branches. Each branch in the CFG is reproduced in the function to determine the next state in the corresponding FSM. This depends upon a condition which is the result of a test (e.g., an arithmetic comparison or a Boolean operation) evaluated in the data path (either true or false). TAO thwarts identification of the correct control flow (i.e., `true` and `false` branches) by assigning a working key bit K_j to each branch j and changing the corresponding test in the controller to be of the form:

$$\texttt{test} \oplus K_j == 1'b1 \tag{4}$$

Based on the key bit K_j, the two branches are reordered to reproduce the correct control flow. For instance, the `true` and `false` blocks are swapped when $K_j = 1$ because the `xor` operation inverts the value of the variable `test`. In this way, the attacker cannot determine which is the actual `true` (`false`) block without knowing the value of the key bit. Fig. 3 shows this transformation on a simple example.

Example. Consider the `if-then` statement in the black box shown in Fig. 3. 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. 3. 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, TAO solutions can yield alternative versions of the controller (shown in the blue boxes in Fig. 3). The two resulting tests are perfectly equivalent, but 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 complex branch constructs such as the `switch-case` by using more working key bits.

3.3.4 Rescheduling Obfuscates Resource Usage. To disguise the arithmetic operations performed in the data path, TAO creates several DFG variations for each basic block. TAO schedules each basic block to determine the number and types of functional units and registers, along with the clock cycle latency, to perform the corresponding computation. This information is then used as constraints for all variations. Algorithm 1 shows the procedure to create the

ALGORITHM 1: TAO 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 is the key bits assigned to BB_i
  Result: $VDFG_i$ is the set of DFG variants associated with BB_i
  $Variants \leftarrow \emptyset$
  $V \leftarrow$ `ComputeKeyVariants`(k_i)
  foreach $v \in V$ **do**
    $dist_v \leftarrow$ `ComputeDistance`(v, k_i) // compute distance between v and k_i
    $DFG*_i \leftarrow$ `CopyDFG`(DFG_i) // create a copy of the current DFG
    $OP \leftarrow$ `ClusterOperations`($DFG*_i$)
    foreach $op \in OP$ **do**
      $op_j \leftarrow$ `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 $dep \in DFG*_i$ **do**
      $dep_j \leftarrow$ `GetDependence`($d, dist_v$) // return a dependence at distance $dist_v$
      `RearrangeDependence`(dep, dep_j) // statistically reorganize the dependences
    end
    $Variants \leftarrow Variants \cup DFG*_i$
  end
  return *Allocation*

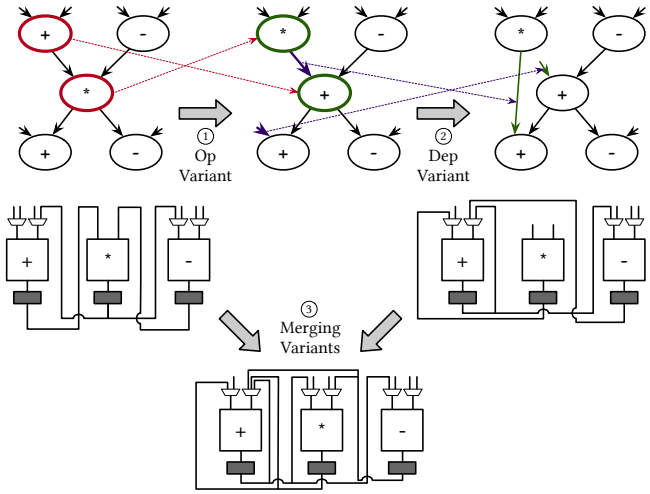

Figure 4: Generation of DFG variants in TAO.

set $Variants$ of DFG variations starting from a valid schedule DFG_i and the key bits assigned to the basic block. Fig. 4 shows the application of this algorithm to a simple example. First, TAO produces the 2^{B_i-1} key variants beginning from the allocated key bits k_i. Then, TAO produces a copy of the current schedule, topologically orders the operations and clusters them based on the operation types. For each operation, TAO determines a reciprocal one in an alternative cluster, and swaps the two operation types with a probability of 0.5 (step ① in Fig. 4). For every DFG edge, TAO elects an alternative edge, and restructures the dependencies to return a credible DFG (step ② in Fig. 4). TAO combines all these solutions into a single data path microarchitecture and restructures the interconnections using extra multiplexers and control signals (step ③ in Fig. 4). 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).

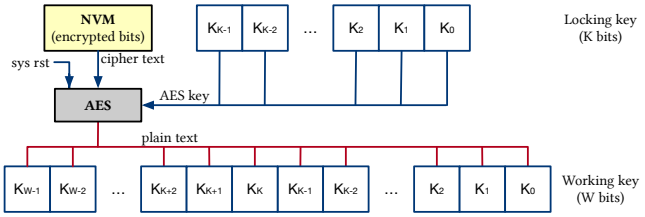

Figure 5: Key management when working key is larger than the locking key ($W > K$). The key is generated at power up and AES decrypts the values in the non-volatile memory.

3.4 Key Management

One feature of TAO is to take care of the *locking key*, i.e., how one delivers it to the IC and generates the *working key*. The locking key is stored in a tamper-proof memory (e.g., EEPROM or Non-Volatile Memory [3]) after IC fabrication [13, 14]. The technology defines the number of locking key bits that one can deliver to the IC. The working key has, instead, an arbitrary size because it depends on the complexity of the algorithm and protection (i.e., number and size of the basic blocks, the number of control branches, and the number of constants).

When we have 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. In this situation, each key bit has a maximum fan-out of $f = \lceil W/K \rceil$, which may compromise the security of the generated IC for large values of f. If the attacker can extract one working key bit, the corresponding locking key bit and all its replicas can be extracted.

TAO proposes an alternative solution shown in Fig. 5. TAO uses the locking key as an AES key to encrypt the working key at design time. Non-Volatile Memory (NVM) in the IC stores the resulting values. At power-up, the values in the NVM are decrypted using the given locking key and loaded into the working-key registers. This solution leverages the security guarantees of a 256-bit AES by using a 256-bit locking key to secure the working key bits.

4 EXPERIMENTAL EVALUATION

To validate our approach, we extended Bambu (ver. 0.9.5) [12], an open-source HLS framework. The modular organization of Bambu enabled us to implement TAO as additional steps in the HLS flow.

4.1 Experimental Setup

We use TAO-enhanced Bambu to generate obfuscated circuits on five 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.

Table 1 shows the characteristics of the benchmarks. For each benchmark, we report 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. These benchmarks are bigger than those used to report logic obfuscation. Working at a higher abstraction allows us to obfuscate larger circuits.

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

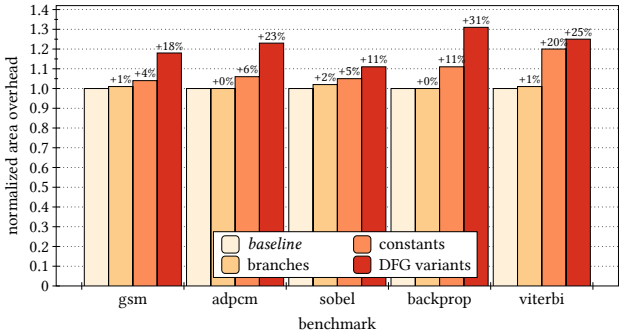

Figure 6: Area overhead of TAO obfuscations.

We set the bit-width of each obfuscated constant to 32 bits (i.e., $C = 32$), while the original constants range between 8 (`char` values) and 32 bits (`int` values). We assign one bit to each control branch. Finally, we assign four bits 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`). We use a 256-bit locking key in all experiments.

To evaluate TAO, we used Bambu HLS targeting the Synopsys SAED 32nm Generic Library at 500 MHz. We synthesized the baseline and obfuscated versions of the circuits using Synopsys Design Compiler J-2014.09-SP2. 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. We extended the testbenches generated by Bambu to specify different locking keys as input and to verify the implementation for each of them. Simulations are performed with Mentor ModelSim SE 10.3 and are instrumented to report if the execution is correct and the number of cycles.

4.2 Overhead

To evaluate the impact of each obfuscation, we modified Bambu to select the methods to apply through command-line options. Since these transformations are orthogonal, we generated different obfuscated versions of the circuits by selectively invoking them. We performed RTL simulations to check the latency regarding 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 create DFG variants in the data path because of the 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, minimally changing the critical path (around 4%). This is proportional to the difference from the actual bits needed to represent the constants.

We performed logic synthesis on the circuits to evaluate the area overhead of the various obfuscations. Fig. 6 shows the results, where each value is normalized against the area of the respective *baseline* version. The results indicate that obfuscating the control branches 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. The creation of DFG variants 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.

We evaluated the overhead of the two key management solutions offered in Section 3.4. 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. Since key decryption is performed only once at power-up, the performance overhead is unimportant once the chip is ready to use.

4.3 Validation of Obfuscation Results

For each benchmark, we randomly generated 100 256-bit locking keys. One key is supplied as input for TAO, while we tested the security level of the created circuit with the others. First, we simulated the generated circuits with the correct locking key corroborating that the circuits produce the correct results. All other keys result in wrong results and this assures that the attacker cannot turn on the circuit with another key. More explicitly, we tested the "output corruptibility" of each locked circuit, computed as the Hamming distance with respect to the output of the baseline circuit [18]. When combined, the three obfuscation techniques produce an average HD of 62.2% over the five benchmarks. Also, 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 in [13], constants and control branches cannot be weakened even with SAT-based attacks. This is because the oracle chip is unavailable in the untrusted foundry threat model. Moreover, the information is fully cut out from the data path and the controller, and one cannot recover it without the correct locking key. The circuits generated by TAO have a higher security level than previous obfuscation techniques at the logic level.

5 CONCLUSIONS AND FUTURE WORK

TAO is a comprehensive solution for algorithmic obfuscation during high-level synthesis. This approach starts from a high-level description of the algorithm and creates a version of the corresponding IC by masking all relevant details through an input locking key. TAO presents a collection of techniques for obfuscating constant values, arithmetic operations, and control branches. TAO implements this comprehensive solution within a state-of-the-art HLS tool and validated on a set of representative benchmarks. These techniques do not incur performance overhead and have an area overhead of around 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] E. Charbon. 1998. Hierarchical watermarking in IC design. In *Proceedings of CICC*. 295–298.
[2] G. De Micheli. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill.
[3] D. Forte, D. Bhunia, and M.M. Tehranipoor. 2017. *Hardware Protection Through Obfuscation*. Springer Publishing Company, Incorporated.
[4] B. L. Gal, C. Andriamisaina, and E. Casseau. 2006. Bit-Width Aware High-Level Synthesis for Digital Signal Processing Systems. In *Proceedings of SOCC*. 175–178.
[5] U. Guin, K. Huang, D. DiMase, J. M. Carulli, M. Tehranipoor, and Y. Makris. 2014. Counterfeit Integrated Circuits: A Rising Threat in the Global Semiconductor Supply Chain. *Proc. IEEE* 102, 8 (Aug. 2014), 1207–1228.
[6] S. Heck, S. Kaza, and D. Pinner. 2011. Creating value in the semiconductor industry. *McKinsey on Semiconductors* (Oct. 2011), 5–144.
[7] J. Hurtarte, E. Wolsheimer, and L. Tafoya. 2007. *Understanding Fabless IC Technology*. Elsevier. 296 pages.
[8] F. Imeson, A. Emtenan, S. Garg, and M.V. Tripunitara. 2013. Securing Computer Hardware Using 3D Integrated Circuit (IC) Technology and Split Manufacturing for Obfuscation. In *Proceedings of SEC*. 495–510.
[9] Y. Lao and K. K. Parhi. 2015. Obfuscating DSP Circuits via High-Level Transformations. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 23, 5 (May 2015), 819–830.
[10] Y. W. Lee and N. A. Touba. 2015. Improving logic obfuscation via logic cone analysis. In *Proceedings of LATS*. 1–6.
[11] R. Nane et al. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Trans. on CAD of Integrated Circuits and Systems* 35, 10 (Oct. 2016).
[12] C. Pilato and F. Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *Proc. of FPL*. 1–4.
[13] J. Rajendran, A. Ali, O. Sinanoglu, and R. Karri. 2015. Belling the CAD: Toward Security-Centric Electronic System Design. *IEEE Trans. on CAD of Integrated Circuits and Systems* 34, 11 (Nov. 2015), 1756–1769.
[14] J. A. Roy, F. Koushanfar, and I. L. Markov. 2010. Ending Piracy of Integrated Circuits. *Computer* 43, 10 (Oct. 2010), 30–38.
[15] L. Stok. 1994. Data Path Synthesis. *Integr. VLSI J.* 18, 1 (1994), 1–71.
[16] P. Subramanyan, S. Ray, and S. Malik. 2015. Evaluating the security of logic encryption algorithms. In *Proceedings of HOST*. 137–143.
[17] V. van der Leest and P. Tuyls. 2013. Anti-counterfeiting with hardware intrinsic security. In *Proceedings of DATE*. 1137–1142.
[18] Yang Xie and Ankur Srivastava. 2016. *Mitigating SAT Attack on Logic Locking*. 127–146.