Hidden in Plaintext: An Obfuscation-based Countermeasure against FPGA Bitstream Tampering Attacks

TAMZIDUL HOQUE, KAI YANG, ROBERT KARAM, SHAHIN TAJIK, DOMENIC FORTE, MARK TEHRANIPOOR, and SWARUP BHUNIA, University of Florida, USA

Field Programmable Gate Arrays (FPGAs) have become an attractive choice for diverse applications due to their reconfigurability and unique security features. However, designs mapped to FPGAs are prone to malicious modifications or tampering of critical functions. Besides, targeted modifications have demonstrably compromised FPGA implementations of various cryptographic primitives. Existing security measures based on encryption and authentication can be bypassed using their side-channel vulnerabilities to execute bitstream tampering attacks. Furthermore, numerous resource-constrained applications are now equipped with low-end FPGAs, which may not support power-hungry cryptographic solutions. In this article, we propose a novel obfuscation-based approach to achieve strong resistance against both random and targeted pre-configuration tampering of critical functions in an FPGA design. Our solution first identifies the unique structural and functional features that separate the critical function from the rest of the design using a machine learning guided framework. The selected features are eliminated by applying appropriate obfuscation techniques, many of which take advantage of “FPGA dark silicon”—unused lookup table resources—to mask the critical functions. Furthermore, following the same obfuscation principle, a redundancy-based technique is proposed to thwart targeted, rule-based, and random tampering. We have developed a complete methodology and custom software toolflow that integrates with commercial tools. By applying the masking technique on a design containing AES, we show the effectiveness of the proposed framework in hiding the critical S-Box function. We implement the redundancy integrated solution in various cryptographic designs to analyze the overhead. To protect 16.2% critical component of a design, the proposed approach incurs an average area overhead of only 2.4% over similar redundancy-based approaches, while achieving strong security.

CCS Concepts: • **Security and privacy** → **Tamper-proof and tamper-resistant designs**;

Additional Key Words and Phrases: FPGA bitstream tampering, FPGA security, Trojan prevention

ACM Reference format:

Tamzidul Hoque, Kai Yang, Robert Karam, Shahin Tajik, Domenic Forte, Mark Tehranipoor, and Swarup Bhunia. 2019. Hidden in Plaintext: An Obfuscation-based Countermeasure against FPGA Bitstream Tampering Attacks. *ACM Trans. Des. Autom. Electron. Syst.* 25, 1, Article 4 (November 2019), 8 pages. <https://doi.org/10.1145/3361147>

1 INTRODUCTION

FPGAs provide a popular reconfigurable platform for accomplishing in-field programmability. Computing systems, which utilize FPGAs benefit from the accelerated operation, generally use

This work is funded in part by Cisco and National Science Foundation (NSF) Grant #1603483.

Authors' address: T. Hoque, K. Yang, R. Karam, S. Tajik, D. Forte, M. Tehranipoor, and S. Bhunia, University of Florida, 968 Center Drive, Gainesville, FL, 32611, USA; emails: {thoque, kyang84, robkaram, stjajik}@ufl.edu, {dforte, tehranipoor, swarup}@ece.ufl.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1084-4309/2019/11-ART4 \$15.00

<https://doi.org/10.1145/3361147>

Fig. 1. The lifecycle of an FPGA-based system is composed of untrusted entities who could involve in or facilitate the tampering of bitstream.

significantly less power than software-based systems. Therefore, these devices have found applications in various domains, including automotive, networking, defense, and consumer electronics. From a security perspective, designs compiled for FPGAs are less vulnerable to certain supply-chain attacks when compared to Application-Specific Integrated Circuits (ASICs), since the design and functionality of the valuable Intellectual Property (IP) are not exposed to untrusted foundries. Therefore, FPGAs are now commonly being used for implementing critical security functions in diverse applications.

However, the commercial development lifecycle of an FPGA-based product exposes its configuration file, or bitstream, to a number of untrusted parties as shown in Figure 1. A malicious entity at the system assembly or deployment facility could cause a deliberate modification of the configuration bits to leak information or cause malfunction [Trimberger and Moore 2014]. For instance, several IoT and automotive systems rely on FPGAs as the root of trust through which images to reprogram every other sub-module is decrypted and authenticated [Al-Omary et al. 2018; Times 2018]. By compromising the FPGA-mapped root of trust IP, the attacker could potentially execute their software and sell the product under a different brand name. Figure 2(a) illustrates the example of a self-authenticating boot process where an authentication module is part of the FPGA-mapped IP to ensure that the IP itself is not tampered before loading onto the device. However, as shown in Figure 2(b), the pass/fail response that either halts the system or permits it to proceed after checking the authentication result could be forced to a desired value by flipping a single bit onto a specific Look-Up Table (LUT).

The absence of bitstream encryption enables a static analysis of the configuration. By iteratively mapping known functions, bitstream format can be partially extracted to observe the configuration of each LUT [Swierczynski et al. 2015a]. Analysis of LUT contents facilitates the identification of the critical function within the bitstream and, hence, significantly reduces the required number of attempts to compromise the system. Furthermore, if bitstream reverse engineering is supported, then the attacker could execute more effective search methods combining various structural and functional features of the target logic within the recovered netlist. Similar analysis has been used to compromise standalone implementations of AES and DES on FPGA through the modification of critical modules within them (e.g., S-Boxes and control logic) [Swierczynski et al. 2018, 2015a].

Bitstream encryption raises the difficulty of modifying a specific function within the bitstream, since a static analysis of encrypted bitstream is not possible. However, encryption naturally requires a decryption module, resulting in a significant area and power overhead. Moreover, unless authentication is used, encryption alone does not secure the bitstream against random and rule-based [Swierczynski et al. 2018] bit flips that could eventually compromise a critical function with a large number of attempts. For many FPGAs with built-in ASIC authentication block (e.g., HMAC) along with a decryption unit, the authentication key and the bitstream are encrypted together with one encryption key. If the attacker gets hold of the symmetric encryption

Fig. 2. (a) An example of bitstream tampering in the field: An IP initiating the boot process authenticates itself at startup to ensure self-integrity, and (b) by altering a specific bit, the *proceed* signal is forced to ignore the authentication result.

key, then the authentication key can be decrypted with the bitstream, and a new digest can be generated from the tampered bitstream to pass the authentication. The encryption key could be leaked through side-channel attacks [Lohrke et al. 2018; Moradi et al. 2013], attacks during remote upgrades, and from an untrusted facility assigned to store the key onto the FPGA. The plaintext bitstream itself could be leaked via contactless probing [Tajik et al. 2017]. Additionally, in low-power embedded applications using FPGAs, the presence of a built-in decryption and authentication block could be infeasible.

In this context, novel, security-aware techniques for tamper *prevention* and *tolerance* are required for continued and improved device security, especially in IoT during long field life. While prevention refers to an increased difficulty of identifying critical functions within the design, tolerance indicates hindering an attacker from attaining a desired malicious outcome under both random and rule-based bitstream alterations. Together, these two features could be termed as tamper-resistance. In this article, we present an approach to enable such resistance in critical functions against static analysis and tampering prior to configuration. We obfuscate both critical and parts of non-critical areas of the design in a way that the discernible functional (e.g., features of Boolean function) and structural properties (e.g., fan-in, fanout) among them are impacted. Hence, static analysis of the bitstream, as well as its netlist, is no longer effective in identifying the critical function. Figure 3(a) shows a partitioned design where the critical function consists of multiple LUTs that are distinguishable from all other LUTs. The uniqueness could stem from some functional features such as the nature of the LUT contents representing their Boolean functions or structural features like fan-in and fanout of the LUTs. As illustrated in Figure 3(b), the initial obfuscation process alters the critical portion to infuse properties from non-critical areas and, similarly, the non-critical regions are modified to have features analogous to the critical logic. By taking advantage of the FPGA dark silicon [Karam et al. 2016] such transformations can be accommodated with low area and power overhead.

Even if the critical function is not distinguishable or static analysis renders very high false positives due to the initial obfuscation process, a motivated adversary could still try to iteratively alter individual candidate LUT at a time to observe the desired malicious outcome. Such alteration can be applied even in the encrypted bitstream [Swierczynski et al. 2018]. However, this attack is feasible if the malicious outcome depends on the alteration of only one LUT (or all LUTs within the same FPGA slice [Swierczynski et al. 2018]), as all possible alteration of multiple LUTs would be difficult to cover. To prevent an attacker from achieving a specific detrimental outcome, we integrate multiple *obfuscated variants* of the critical function. As shown in Figure 3(c), the target function is transformed into three different variants that appear structurally and functionally different from each other. These variants pass their responses to an Obfuscated Voting Unit (OVU) that integrates the output from individual variants of the critical function (Figure 3(d)). Hence, to achieve the intended failure, the attacker is required to alter at least a majority of the variants that lead to a significant increase in attack space. However, instead of altering any variants, the

Fig. 3. Overview of the proposed technique: (a) A critical function with distinguishable properties compared to rest of the design; (b) critical and non-critical functions are obfuscated by infusing properties from their counterparts to ensure indistinguishability; (c) to prevent random and rule-based modifications, the critical function is transformed to three functional variants integrated through voting; and (d) a voting monitoring unit is also integrated to detect any tampering of the voting logic.

adversary could try to modify the OVU only. Therefore, to protect the OVU, a Voting Monitoring Unit (VMU) is integrated to ensure that the whole design becomes non-functional when an OVU alteration is detected. By corrupting the output of the whole design, the VMU ensures that the specific malicious outcome desired by the attacker cannot be achieved through OVU alteration only. Hence, now the attacker must tamper multiple different logic units simultaneously (i.e., either multiple variants or OVU-VMU pairs) to achieve the intended malicious impact without halting the system. The proposed countermeasure offers the following key benefits:

- It does not require any architectural modification to the FPGA. It can, therefore, be used with existing devices simply by updating their configuration files.
- It is not vendor-specific and requires minor modification to the existing synthesis flows.
- It provides significant tamper-resistance of the critical functions without requiring bit-stream encryption and authentication.

As we demonstrate in our security analysis, the problem of identifying and isolating obfuscated critical function grows significantly, and, hence, the technique is highly resistant to known design or template matching attacks. In particular, we make the following contributions in this article:

- We propose an obfuscation-based mechanism for bitstream tamper-resistance, giving FPGAs the ability to protect against malicious alteration of security-critical functions specified by the IP designer. We show that the proposed framework can take advantage of FPGA dark silicon to render the critical function indistinguishable while generating multiple obfuscated variants of the function to improve its tamper-tolerance in an area-efficient manner.

- We present a complete methodology and a custom CAD flow, which seamlessly interfaces with the commercial toolchain. The tool integrates with Synopsys Synplify Pro FPGA synthesis software [Pro 2018], and therefore, it can be used with FPGAs from all major vendors, including Xilinx and Intel.
- To evaluate the quality of obfuscation, we demonstrate a machine learning-based critical function identification method. We assess our technique in identifying AES S-Boxes in a proof-of-concept design containing an AES core, a DLX processor, and an FFT module.
- We analyze the area, power, and delay overhead of integrating three obfuscated variants following our approach with comparable techniques (i.e., triple modular redundancy) and show that the proposed technique enables significantly higher resistance against tampering with the cost of minor overhead.

Section 2 describes the bitstream tampering threat model with the limitations of existing solutions and provides an in-depth look at the prevalence of unused LUT inputs in post-synthesis designs. Section 3 describes the tamper resistance methodology, while Section 4 describes the software toolflow. Section 5 presents the performance and area overhead and analyzes the security of the approach. We conclude in Section 6.

2 BACKGROUND

In this section, the applicability of the proposed approach is discussed. A brief discussion on the FPGA dark silicon is also presented.

2.1 Threat Model: Bitstream Tampering

2.1.1 Adversary and Trust Model. Someone with malicious intent at the assembly, test, or deployment facility with complete physical access to the FPGA-based system could be an adversary. The attacker can obtain the bitstream from the non-volatile configuration storage, alter as required, and rewrite the tampered bitstream. During operation, the attacker can provide inputs to the system and observe the outputs as a black-box. The attacker can locate the LUT configurations by generating several example bitstream files for a number of known functions (e.g., AND, OR, XOR) and analyzing them. She can also obtain the netlist from the target bitstream using bitstream reverse engineering (RE) tools [Pham et al. 2017]. While almost all RE tools are vendor and device specific, we still consider such a scenario, since applying static analysis on a netlist is far more feasible than applying it to a bitstream. In addition to the LUT configuration, various structural properties of the target function can be used at the netlist level for the identification of the critical function. By observing a number of publicly available implementations of a function, the attacker can learn the functional and structural features that could make it distinguishable from the rest of the design.

We assume that the attacker does not have knowledge about most of the non-critical parts of the design and does not know the exact physical location (distance from primary input/output) of the critical function within the IP.

2.1.2 Attack Methods. Several methods exist to cause modifications to an FPGA configuration file. As shown in Figure 4, alteration could be carried out at runtime through fault injection attacks [Skorobogatov and Anderson 2002] or by triggering of a hardware Trojan implanted on the physical device during fabrication [Mal-Sarkar et al. 2016]. The device could even be reprogrammed with an entirely different bitstream by an attacker, possibly during a remote upgrade via spoofing attacks [Trimberger and Moore 2014]. In this work, we focus on direct alterations of the critical function of a design prior to programming. More specifically, we consider the following attack methods.

Fig. 4. Classification of tampering attacks on FPGA bitstream. The proposed countermeasure primarily focuses on modification of critical function prior to configuration.

Targeted Modification (TM): Targeted modifications can be applied to both bitstream and netlist representation of a design. The idea is to reduce the number of LUTs to be altered with the help of static analysis. In Swierczynski et al. [2015a], a generic mechanism to identify the LUT configurations in a bitstream through an iterative compilation of known functions has been demonstrated. Based on subsequent analysis of only one feature related to the non-linearity of LUT configuration, S-Boxes were identified in the bitstream of an AES. Finally, by only modifying the identified S-Box LUTs, a specific malicious outcome was observed. The non-linearity $N(f_k)$ of a Boolean function is calculated from the maximum unexpected distance obtained from the Walsh spectrum. For a k input Boolean function f_k , if the Walsh spectrum is denoted as $W(f_k)$, then the maximum unexpected distance (MaxUD) is found as $\max_{a \in F_{2k}} |W(a)|$. The non-linearity $N(f_k)$ is just a function of input size k and MaxUD [Sosa 2016]:

$$N(f_k) = 2^{k-1} - \frac{1}{2} \max_{a \in F_{2k}} |W(a)|. \quad (1)$$

In the equation above, a_k is an affine function. We also represent entropy of the LUT configuration based on Shannon entropy, which could be useful in identifying a function with an expected ratio of 0s and 1s (e.g., AND/OR/XOR function). If the probability of 1 and 0 in the LUT configuration is P_1 and $(1 - P_1)$, respectively, then we define the entropy as

$$H = P_1 \log_2 \frac{1}{P_1} + (1 - P_1) \log_2 \frac{1}{(1 - P_1)}. \quad (2)$$

Static analysis is more feasible at the netlist level where the contents of all FPGA resources and their connections can be observed. While obfuscating the LUT configuration could make it difficult to identify a known function, the inclusion of structural information from netlist (fan-in and fanout of LUTs) could still make the function distinguishable and facilitate targeted modification.

Random Modification (RM): An attacker can try to flip bits randomly without even knowing the bitstream format. For instance, if the goal is to bypass a specific restriction implemented through a single signal that provides a yes or no response, then it could be a matter of flipping a single bit that eventually goes to the output of the driving LUT. Even in the presence of encryption, the bitstream is vulnerable to such bit-flips. Xilinx uses AES-256 in Cipher Block Chaining (CBC) mode, and Intel uses AES-256 in Counter mode (CTR). In both modes, a localized bit-flip in the plaintext is possible by altering the ciphertext [Trimberger and Moore 2014].

Rule-Based Modification (RBM): Xilinx FPGAs can be viewed as arrays of slices where each slice contains multiple LUTs. A rule-based modification of bitstream in the granularity of LUTs or slices (e.g., four LUTs) requires significantly less number of attempts compared to modification in bit-level granularity [Swierczynski et al. 2018]. Such alteration has been used for compromising LUT implemented AES controller logic within a few hours. The key advantage of such an attack

is that it does not require prior knowledge of the critical function and can even be applied to an encrypted bitstream (with a higher number of attempts). However, for unencrypted bitstream, the attack becomes more powerful if distinguishable features of the critical function are known.

2.1.3 Assumptions. We consider a scenario where bitstream encryption and authentication is either not available or the decryption key is compromised such that the design can be altered in decrypted form. For instance, bitstream authentication is not present in Xilinx Spartan 6, Virtex II, Virtex 4, and Virtex 5 series devices and the encryption key has been compromised through side-channel attacks on these devices [Swierczynski 2018]. Apart from the known critical function, the entire design should contain unknown functional modules. This is a common scenario as the entire FPGA is likely to contain several functional units (e.g., processor, crypto core, DSP, etc.) responsible for different activities. One real-world example of such a multi-module design would be the automotive infotainment companion chip platform that is built on Spartan-6 FPGA [Xilinx 2019]. Preventing S-Box tampering in a standalone AES is difficult if the netlist is available using reverse engineering. However, at this point, the attacker does not even have to tamper with the original AES. Instead, the attacker could merely compromise a known AES design (e.g., one that is being used for learning the critical function) and reprogram the FPGA replacing the actual standalone AES. A more common scenario would be where the AES is just part of a larger FPGA-mapped IP. Such spoofing is not feasible in such a scenario and tampering with the critical function residing in an unknown design is the only option. Finally, we assume that the attacker only wants to make a specific malicious change without making the entire system non-functional. For instance, an attacker trying to bypass an FPGA-implemented authentication of some binaries would want the system to work after the authentication is bypassed. Otherwise, the system is no longer functional for running the desired binaries.

2.2 Existing Solutions

Various effective countermeasures exist to detect and tolerate *accidental* alterations of the bitstream caused by single event upsets (SEUs) [Ebrahimi et al. 2016]. However, these solutions could fall short against *intentional* modifications, since (1) the latter could be done intelligently to cause a desired outcome, (2) successful modifications could have a more catastrophic impact, and (3) countermeasures are easier to bypass by an attacker, since they usually do not go through a comprehensive security analysis that considers various attack models. For instance, the Cyclic Redundancy Check (CRC) offered by Xilinx to check bitstream integrity could be disabled by a single bit-flip within the bitstream, which has already been demonstrated in Chakraborty et al. [2013]. Another Xilinx specific fault mitigation tool called security monitor IP core (SecMon) is designed for detecting and correcting a specific number of simultaneous faults at runtime. However, it is primarily made available to defense contractors and government agencies [Xilinx 2015]. As already mentioned, encryption and authentication-based solutions are vulnerable to key leakage and may not be suitable for resource constraint applications that rely on low-end FPGAs. In Swierczynski et al. [2015b], a technique to hide the AES S-Box using dynamic reconfiguration has been proposed. At design time, the original S-Box is replaced with a modified one, and after configuration, the S-Box is updated through partial reconfiguration. Therefore, by analyzing the bitstream before configuration, the attacker cannot identify the S-Box using known features. However, along with the partial reconfiguration feature, this solution requires the presence of CFGLUT5 primitives. Furthermore, the solution is only applicable to cryptographic IP cores or requires encryption/decryption support, as the correct configuration of the critical functions has to be stored within the design in encrypted form. By introducing architectural modification in FPGAs, Karam et al. [2017] tries to impede tampering by generating a node-locked bitstream for each device. Unlike existing

Fig. 5. Mapping results from the IP cores in the IWLS Benchmark Suite.

solutions, our proposed technique does not require cryptographic support, partial reconfiguration feature, or architectural modifications to the FPGA. Rather, the method is applicable to any FPGA present in the market, irrespective of vendor and FPGA series.

2.3 FPGA Dark Silicon

A previous study into FPGA dark silicon [Karam et al. 2016] has demonstrated that for various benchmarks and IP cores mapped to an Intel Cyclone V FPGA, the vast majority of “used” resources are not, in fact, fully utilized in practice. For example, a modern FPGA may support a six-input function in hardware lookup tables, but in reality, a very small percentage of LUTs in the design actually make use of all six inputs. This leaves a large amount of empty space in the “used” portion of the design, which can be leveraged for integrating additional key input(s) for obfuscation.

We present an additional investigation into the phenomenon of FPGA dark silicon by mapping 16 different IP cores from IWLS benchmark suite [Albrecht 2005] in Intel (formerly Altera) MAX10, Cyclone V, and Stratix IV FPGAs. Also, we present a comparison of how the distribution of LUTs varies between performance-optimized and area-optimized designs, noting that area-optimized generally uses fewer resources overall. Figure 5 shows these mapping results. The MAX10 FPGA supports up to four input functions, which generally constitute 50% of the LUTs in the post-synthesis design. Cyclone V and Stratix IV FPGAs have similar results, with a small fraction of the LUTs making use of the seven-input “extended mode” configuration, roughly 28% (performance-optimized) and 19% (area-optimized) LUTs contain a maximum of six inputs, leaving roughly 70% (performance-optimized) and 80% (area-optimized) LUTs with 1 or more input free. These can be used for the addition of one or more key bits, or in this case, for the obfuscation of critical and non-critical functions. However, in the bitstream obfuscation method proposed in Karam et al. [2016], the unused contents were filled with random values after key insertion, which did not provide the maximum benefit for hiding a critical function. This is common in many other obfuscation techniques that provide functional locking but fail to cause significant structural alteration to hide the design intent [Chakraborty et al. 2018, 2019]. In this article, we ensure that the contents being inserted in unused LUT space reduce the differences among critical and non-critical segments.

Fig. 6. (a) The predefined critical function of a design is first analyzed to identify the possible unique features it may have. (b) During the variant generation process, appropriate transformations are applied to the critical function such that the unique features are not present in the generated variants.

3 METHODOLOGY

3.1 Architecture and Operation

The proposed approach consists of five integral modules. The first four, as described in Figure 3, include the original design with parts of non-critical functions modified, three variants of the critical function, OVUs for majority voting, and VMUs for monitoring the voting. There is also a finite state machine for applying the obfuscation key and observing responses from VMUs. Below we describe each of these modules.

3.1.1 Variants of the Critical Function. The critical function is primarily viewed as a segment of the whole design containing LUTs and flip-flops (FFs). Before applying any transformation to the design, each LUT of the critical function is assessed for the possible identifying features that have to be eliminated. As we discuss later in detail, various obfuscation methods are presented that are capable of eliminating a number of known features. The selected transformations are applied to generate three variants of the critical function based on a random key. The variants provide the same functionality as the original one upon the application of the correct obfuscation key during operation. The generated variants are similar to different non-critical segments of the design in a varying degree. Therefore, the variants are not identical to the attacker-known critical function and appear different from each other as well. Each variant would contain a varying number of LUTs and key inputs. The configuration of comparable LUTs across variants and their fan-in/fanout will be diverse as well. Following the same approach, areas of non-critical segments are obfuscated to infuse the known features of the critical function. The exact process of identifying and manipulating these features is described in Section 3.2. To illustrate the concept more clearly, Figure 6(a), shows an FPGA-mapped netlist of a design with a critical function that compares the function's input to an 8-bit value (i.e., 8'h00) and asserts logic high if a match is found. The critical function contains two LUTs named LUT_First and LUT_Second. Their configuration bits (i.e., 16'hFEE and 32'h00000001) contain an imbalanced number of 1s and 0s in each that contributes to lowering the entropy of configurations. This makes them distinguishable assuming that the non-critical LUTs in the design do not have such low entropy in their contents. Therefore, when the three variants are generated, transformations are applied to LUT_First and LUT_Second such that their entropy increases. Figure 6(b) shows how integrating bits to balance 1s and 0s (marked red) in the critical LUTs through key-insertion increases the entropy of the three variants. Hence, the critical LUTs can no longer be distinguished from the rest of the design by analyzing the entropy. To further increase the complexity for the attacker, various non-critical LUTs with high entropy in the original design could be transformed to have low entropy.

Fig. 7. An FSM is implemented to apply the obfuscation key and monitor the responses from all VMUs. If VMU responses alter, then the FSM moves to a locked state where the design produces corrupted output.

Once the variants are generated, they are integrated with OVUs and VMUs to replace the original critical function. As shown in Figure 7, each of the three variants C_1, C_2, C_3 is receiving the original input $A[0 : a]$ to the function. The obfuscation key $K[0 : k]$ is also distributed across the variants. Here we assume that the original function has one output $O[x]$, which now is generated by the majority voting in OVU. In practice, the critical function could have any number of inputs and outputs. For each output signal, a corresponding OVU and VMU has to be integrated.

3.1.2 OVU and VMU. Each comparable triplet of outputs from variants (i.e., $C1_{out}, C2_{out}, C3_{out}$) is fed to an OVU, which generates a response (i.e., $O[x]$) based on the majority vote of inputs. This response is then passed to the appropriate non-critical segments. At this point, the OVU is identical to a simple majority voting function implemented as: $O[x] = (C1_{out} \wedge C2_{out}) \vee (C1_{out} \wedge C3_{out}) \vee (C2_{out} \wedge C3_{out})$. Since this logic could be searched with a known configuration signature and other structural properties, we integrate various mechanisms to protect it. We propose the following possible measures to increase the difficulty of identifying OVU.

- For an FPGA with six-input LUTs, each OVU uses a single LUT with three unused inputs. This could be utilized to incorporate three different key inputs that provide a large number of possible transformations. Transformations are applied to obfuscate the LUT configuration and structure (fan-in, fanout) of OVU.
- Some of the variants could be configured to provide inverted responses that would help to obfuscate the Boolean function of OVU further when the inverted relations are taken into consideration while generating the voting logic.
- Variants of OVU-like contents could be inserted in dark silicon areas of non-OVU LUTs to increase false detection during OVU search.

While the above-mentioned techniques help to hide the OVU, an attacker can still apply random and rule-based alterations to impact the OVU. To thwart such modification, a VMU is connected to each OVU as the last line of defense. The three comparable outputs from variants (i.e., $C1_{out}, C2_{out}, C3_{out}$) along with the corresponding output of OVU (i.e., $O[x]$) is fed to the VMU as inputs. To check OVU's integrity, the VMU performs the same voting and compares the result with $O[x]$. One possible implementation of such logic would be: $M[x] = O[x] \oplus ((C1_{out} \wedge C2_{out}) \vee (C1_{out} \wedge C3_{out}) \vee (C2_{out} \wedge C3_{out}))$. The signal state of $M[x]$ that indicates tampering could be 0 or 1 depending on the logic used for comparison (e.g., XOR/XNOR).

The VMU output feeds to a key application and monitoring FSM (KAM) that is responsible for providing the obfuscation key to different LUTs of the design. As soon as tampering is detected, the KAM provides an invalid key to ensure functional failure. Therefore, the benefit of successfully attacking the OVU would likely be eliminated. For instance, if the attacker is modifying the control logic of AES to observe a faulty ciphertext with a specific property, then rendering the design non-functional after detecting the tampering would provide a different faulty ciphertext that is of no use to the attacker. By integrating VMU, we are significantly increasing resistance against random and rule-based alterations as the compromise would require simultaneous modification of both OVU and VMU.

The OVU and VMU should not be placed in the same slice, as it may still allow an attacker to compromise both by altering a single slice. The attacker can also alter all slices in a single Configurable Logic Block (CLB) column depending on the FPGA configuration frame structure. Hence, OVU and VMU could be at risk of simultaneous alteration if present within the same CLB column. However, CLB-level alteration is more coarse-grained compared to LUT and slice-level alterations. For a coarse-grained modification, the attacker is more likely to accidentally alter adjacent non-critical portions as well that share a CLB/slice with the critical logic. This could corrupt the system's output in such a way that it circumvents the actual goal of the attacker. Hence, the critical function could be considered safe using the same CLB (or even slice) for the VMU and OVU if other non-critical LUTs with high output corruptibility are placed adjacently. The VMU could be secured further by incorporating some of the obfuscation-based measures mentioned earlier for OVUs.

3.1.3 KAM Operation and Design. The obfuscation key to all obfuscated segments of the design is supplied through KAM. It is also responsible for corrupting the outputs by providing the wrong obfuscation key when a VMU indicates tampering of OVU. As shown in Figure 7, the state-space of KAM is divided into the following set of states:

- *Initial states, $S_{0,1,2,\dots,p}^{IS}$* : During system startup, the KAM commence from the initial state S_0^{IS} and traverses in response to some inputs $F[0 : f]$. These inputs are selected from the internal nets of the IP at design time. During the traversal of these states, different incorrect obfuscation keys are generated by the KAM until the valid key application state S_0^{VK} is reached.
- *Valid key application state, S_0^{VK}* : When the KAM reaches state S_0^{VK} , the correct obfuscation key is applied for the first time, and all obfuscated areas of the design produce correct functionality. This state is marked in green in Figure 7.
- *Valid key application and monitoring states, $S_{1,2,\dots,r}^{VK}$* : Response from $F[0 : f]$ further transitions the KAM from state S_0^{VK} to one of the monitoring states where the correct obfuscation key is kept asserted and responses from VMUs are being observed by the KAM continuously. As long as the VMU responses are correct, the KAM keeps traversing within these monitoring states (yellow states in Figure 7), irrespective of $F[0 : f]$. Switching between multiple monitoring states keeps the state elements of KAM active and reduces the opportunity for KAM logic detection through runtime activity analysis of netlist.
- *Locked states, $S_{0,1,2,\dots,q}^{LS}$* : If VMU responses indicate tampering, then the KAM moves to locked states where invalid obfuscation keys are generated to cause output corruption. There is no transition path to the initial or valid key states from the locked states.

While the idea of key application FSM used here is similar to the one used in state-space obfuscation [Chakraborty and Bhunia 2009], the proposed KAM is traversed using responses from internal signals residing in the unobfuscated non-critical part of the IP instead of primary inputs.

Fig. 8. The overall methodology for implementing the proposed approach.

At the design stage, a set of $f + 1$ internal signals $F[0 : f]$ is selected such that it can traverse the KAM from the initial state S_0^{IK} to one of the valid key application and monitoring states $S_{1...r}^{VK}$. It must be ensured that for all possible expected inputs to the IP until z clock cycle after startup, the sequence of values from $F[0 : f]$ is consistent. Identifying candidate internal nets for $F[0 : f]$ is similar to finding stable signals within a given netlist for generating stealthy opaque predicates in a hardware design that could be used as obfuscated constant signals [Hoffmann and Paar 2018]. A similar concept is used here for identifying $F[0 : f]$ that produces a known sequence of values until a desired time period. In our application, $F[0 : f]$ is chosen such that after startup, the KAM always reaches to state S_p^{IS} from S_0^{IS} within $z - 2$ cycles. Until this period, the KAM is designed to provide varying incorrect key values across different states. In $z - 1$ cycle, $F[0 : f]$ drives the KAM to the first valid key application state S_0^{VK} , where the correct obfuscation key is applied, and the obfuscated areas of the design become functional. Finally, in z th cycle, the design moves to one of the states, where VMU responses are constantly monitored while keeping the valid obfuscation key asserted (i.e., $S_{1,2...r}^{VK}$). In these states, the KAM is no longer sensitive to $F[0 : f]$ and leaves the valid key application states only when the output signals of VMUs, $M[0 : m]$, indicate tampering. In such a situation, KAM reaches to one of the locked states producing an invalid obfuscation key and stays there unless the system is restarted. At the design stage, KAM for a variant integrated IP is constructed based on the obfuscation key to be generated, VMU responses to be monitored, selected input signals to drive KAM, number of valid and invalid key application states to be included, and their state transitions.

3.2 Implementation

The implementation of the proposed method consists of four major steps as shown in Figure 8. First, the critical function within the FPGA-mapped netlist is identified. We assume that the designer of the IP core decides what could be a critical function in the design. A structural netlist [Xilinx 2014] where individual submodules of the IP (e.g., control unit, datapath, instruction memory, etc., in a DLX processor) can be separated could be used to locate the known critical function. In a flattened netlist, it would be difficult to locate the logic areas belonging to the critical function as all the submodules are merged into one common netlist with no visible separation. However, after integrating the required changes, the design can be resynthesized as a flattened netlist. A critical function could be realized as a group of LUTs, FFs, and other FPGA specific primitives. Functions realized as ASIC blocks (e.g., DSP) offer very limited flexibility for modification, which makes them less attractive for tampering. Hence, during obfuscation, we do not alter these modules. After locating the critical function, the successive major steps are followed:

3.2.1 Uniqueness Analysis. Uniqueness analysis helps to understand which parts of the given critical function is distinguishable from the rest of the design using a set of features. For this

Fig. 9. (a) The final result from K -fold cross-validation process can be used for uniqueness analysis; (b) a wrapper-based feature selection technique can be applied to understand the impact of various features in the classification of LUTs.

purpose, the values of a set of predefined features for each LUT of the design (both critical, non-critical) has to be extracted. However, there is no generic list of features applicable to all possible functions. Yet, relevant circuit recognition-based work can be studied for possible features [Hoque et al. 2018]. For S-Box identification in AES, the non-linearity of the LUT configuration has been used, since the S-Box is known to store such values [Swierczynski et al. 2015a]. Some of the example features are already discussed in Section 2.1.

Once the values of the selected features are extracted for each LUT of the IP, the instances are labeled as critical/non-critical. If only one or two features are available, then the uniqueness could be analyzed by observing the distribution of labeled feature values. If distributions for critical and non-critical LUTs for a feature do not overlap, then an attacker could potentially use this feature to identify the critical function [Swierczynski et al. 2015a]. If there is a significant overlap in two distributions, then it indicates that the feature alone is not useful for separating the critical LUTs. The uniqueness analysis step requires the designer to perform similar static analysis to understand which LUTs are easily identifiable.

In the presence of many features, a machine learning-based approach would be more useful compared to a discrete analysis using individual features. A supervised classifier could be trained and tested using the extracted data of features to classify critical and non-critical LUTs. To assess the performance of the trained model, a k -fold cross-validation could be done where the data is randomized such that all of the folds contain some instances from both critical and non-critical classes. As shown in Figure 9(a), in each iteration, one of the folds is used for testing while the rest are used for training the classifier. Each iteration provides classification results based on the corresponding training and test data. Results for all folds are summarized to obtain a final classification result. Classification results can be represented by the following metrics:

- False Positive (FP): Number of non-critical LUTs falsely classified as critical LUTs.
- False Negatives (FN): Number of critical LUTs falsely classified as non-critical LUTs.
- True Positives (TP): Number of critical LUTs correctly classified.
- True Negatives (TN): Number of non-critical LUTs correctly classified.

If the non-critical LUTs are very distinguishable from the critical ones, then the validation will provide high true positives and true negatives, indicating that most LUTs can be classified correctly. Conversely, if the features are not useful, then the number of false positives and false negatives would be high. The designer can also observe which specific LUTs are classified/misclassified. This information would allow the designer to transform the specific LUTs that could be easily classified by an attacker. In Section 5.1, we show the implementation of a similar supervised learning-based technique using a Random Forest classifier to classify the LUTs of the AES S-Box in an FPGA-mapped netlist. Uniqueness analysis could also be performed using the unsupervised

clustering-based machine learning. If the critical LUT instances form a separate cluster, then it indicates that they are vulnerable to easy identification by an attacker.

3.2.2 Transformation Assignment. Based on the results from uniqueness analysis, LUTs are selected for applying various transformations. Critical LUTs that generate false negatives and non-critical LUTs that generate false positives during the uniqueness analysis step do not need any transformation as they are already misclassified. Instead, LUTs that are correctly classified are considered for the transformation assignment step. Since these LUTs can be correctly identified by an adversary as critical or non-critical, appropriate transformations are needed for these LUTs to eliminate their distinguishing features. To understand which transformation is useful for a given LUT, it is necessary to identify the features that are helping the most in correct classification. A wrapper-based feature selection method could be used to understand which features are the most useful in detecting the majority of LUTs [Liu and Yu 2005]. In wrapper algorithm, a classifier (i.e., Random Forest) is predefined first and utilized for feature subset evaluation. As shown in Figure 9(b), for each combination of feature subset, the classifier is applied to data, and the goodness of the subset is determined by assessing the classification results. The final output of the feature selection process is a set of features ranked according to their influence in classifying the LUTs. At this point, we can select transformations to eliminate the influential features from all detected LUTs. For instance, if we observe that the non-linearity of contents is causing most of the LUTs to be correctly classified, then all the detected LUTs could be obfuscated with a key such that the non-linearity values of the transformed LUTs shift towards their opposite class in varying amounts. Similarly, if the high fanout of a LUT is causing them to be detected, then dummy fanout could be added with other LUTs of the opposite class to ensure false detection.

A more precise but time-consuming transformation assignment approach would be to obtain a list of influential features for each correctly classified LUT so that the minimum possible transformation is needed. For that purpose, while applying the wrapper method, the classifier's output for each feature subset could be compared to locate the most influential features for each LUT.

3.2.3 Transformation Techniques. LUT-level transformation using key insertion and shuffling was proposed earlier [Karam et al. 2016]. In this article, we further include fusion and segmentation to demonstrate the proper use of these transformations to eliminate distinguishing functional features. For structural modification, we also suggest the use of the existing MUX insertion technique [Lee and Toubia 2015]. Applications of these transformations are described below with their algorithms presented in Appendix A. With further exploration of unknown features, new transformation techniques can be included to eliminate the features from a given function.

Shuffling: Shuffling of the LUT inputs is the most basic form of obfuscation method. However, placement and routing (PAR) prior to the bitstream generation inherently causes the inputs to shuffle. We still apply shuffling at the netlist-level as it is not guaranteed that the shuffling would occur inherently for all LUTs during PAR. Figure 10(a) illustrates how the shuffling of input changes the bits within the LUT to achieve the same functionality. Input shuffling can achieve a maximum of $X!$ possible transformations for a given X input function mapped onto a single LUT. As shown in Table 1, shuffling only impedes a direct matching of LUT contents.

The process of generating shuffled LUT contents due to input shuffling is described by Algorithm 1 in Appendix A. The truth table of the target LUT can be observed as a two-dimensional (2D) array segmented into multiple input columns and one output column. A shuffling sequence of inputs is chosen randomly (I_0, I_1, I_2, I_3 could be shuffled to I_2, I_1, I_3, I_0). Based on that sequence, the input columns of the original truth table are shuffled. The grouped input columns of the original and shuffled truth tables can be defined as I_{org} and I_{new} 2D arrays, respectively. Each row of I_{org} is searched within I_{new} to find the mapping of rows between these two arrays. The output column

Fig. 10. Different transformation techniques: (a) *input shuffling* is applicable for all LUTs, (b) *key insertion* is applicable to LUTs with unused inputs, (c) *fusion* could be implemented between consecutive LUTs by sharing a common key, and (d) *segmentation* replaces a LUT with three new LUTs divided into two levels.

of the original truth table can be shuffled according to this mapping to generate the output column of the shuffled truth table. The algorithm presented would be repeated for all LUTs that needs to be applied with shuffling transformation. For individual LUTs, the computational time is constant, since the nested for-loop that loops through I_{new} and I_{org} would not require more than $r2 * r1$ number of computations, where $r1$ and $r2$ are the content length of the original and new LUTs respectively (e.g., 64, 32, etc.). However, the runtime would linearly increase with the number of LUTs to be shuffled ($O(n)$ complexity for n LUTs).

Key Insertion: As discussed in Section 2.3, the majority of LUTs in a design are underutilized, and thus, provide unused inputs. Key-dependent obfuscation could be achieved by realizing one or more unused inputs as key input(s). Besides, the key input could also be shuffled with the functional inputs to achieve stronger obfuscation that increases the search space for an attacker. As shown in Figure 10(b), the function $A \oplus B$ is transformed into a three input function that enables the original XOR function when the inserted input K is applied with the correct value 0. In Karam et al. [2016], the unused memory space of the LUT was filled with random values, while in our approach, the selection of the content is generated judiciously to impact particular features. As shown in Table 1, key insertion could potentially impact several features related to the LUT configuration and fan-in.

Algorithm 2 provides a formal approach to execute a key insertion where the key is first inserted as the outermost input for the ease of applying the transformation. As shown in Figure 10(b), any change in key input K inserted as the outermost input selects within the top or bottom half of the truth table. Hence, based on the correct value of the key, either the top or bottom half could be inserted with desired dummy bits. The time required to insert single or multiple keys is constant. The complexity of the whole design is defined as $O(n)$, where n is the number of LUTs to be applied with key insertion.

The key insertion algorithm assumes the dummy bits to be inserted for the key input is either random or predefined. The generation of appropriate dummy bits for candidate LUTs could be done separately. In the proposed framework, dummy bits are calculated such that certain features of the modified LUT are impacted. For instance, to impact a LUT configuration with high entropy through key insertion, dummy contents would be appended such that the entropy reduces after modification. If the equation to calculate the dummy bits for impacting a feature is unknown, then random contents could be iteratively concatenated with the original one and the corresponding feature value could be observed. The loop executes as long as the value of the feature in the transformed LUT does not reach the desired range.

Table 1. Comparison of Transformation Impacts on Various LUT-level Features and Additional LUT(s) Required

Transformation	Sequence of LUT Content	Entropy of LUT Content	Non-linearity of LUT Content	Fan-in of LUT	Fanout of other LUT	Overhead (Additional LUTs)
Shuffling	✓	✗	✗	✗	✗	0
Key Insertion	✓	✓	✓	✓	✗	0
Fusion	✓	✗	✗	✗	✗	0
Segmentation	✓	✓	✗	✓	✓	2x
MUX Insertion	✗	✗	✗	✗	✓	1x

Fusion: Two consecutive LUTs, each having at least one unused input could be keyed together in a way that the preceding LUT can operate in an inverted manner, while the LUT at the front fixes the inversion. As shown in Figure 10(c), a single LUT-mapped function $Z = A \oplus B$ could be modified with a key K to execute a random function (when $K = 0$), and to execute the inverted of the original (when $K = 1$). Since this transformation generates a faulty value of Z in both values of the key, the following LUTs that take Z as input consider an inverted value of Z when $K = 1$. This effectively doubles the possible variations on top of shuffling and key insertion. Similar to shuffling, fusion only impacts the direct matching of LUT contents.

Algorithm 3 shows the technique to apply fusion-based key insertion for a given LUT pair with at least one unused input in each. The LUT pair can even have an FF in between. The rear LUT is first expanded with a key. Unlike key insertion, the correct LUT bits are stored in an inverted form. To accommodate this change, contents of the front LUT needs to be swapped. The swapping distance depends on which input of the front LUT the rear LUT connects to. After the swapping of contents is done, key insertion to the front LUT is applied with the same key as the rear LUT. The time complexity is constant as the for-loop iterates $2^Z/2$ times for a Z input front LUT. All other computations are also constant per LUT.

Segmentation: Segmentation improves the security for designs where dark silicon is not available, or a significant structural obfuscation is desired. A similar function decomposition method has been suggested in Swierczynski [2018], but its effectiveness in hiding a critical function was not demonstrated. Figure 10(d) illustrates the segmentation of a completely utilized LUT. First, one of the inputs (preferably the outermost) of the original LUT is separated (A_6 in the figure). The rest of the inputs (i.e., $A_1 : A_5$) are used to generate two new segmented LUTs. The new LUTs can be shuffled and inserted with keys independently. To regenerate the original functionality, outputs of the two segmented LUTs are driven to a common LUT acting as a MUX. The remaining input A_6 is now connected as the select pin. This common LUT is further obfuscated with shuffling and key insertion, since only three of its inputs are used. The segmentation process described through Algorithm 4 shows that the time complexity is constant as the for-loop present iterates no more than 2^Z times for a Z input candidate LUT.

As mentioned in Table 1, segmentation could be used to impact several structural and functional features. While segmentation helps to modify (reduce) the fan-in of the transformed function, the LUT instantiated 2:1 MUX that connects the two segmented LUTs could be transformed to 4:2 MUX to take two additional inputs from other parts of the design as dummy inputs. This would increase the fanout of other LUTs. Furthermore, it helps to accommodate key insertion to impact the functional features in LUTs with no unused inputs available.

MUX Insertion: MUX insertion is a well-known obfuscation method for IP protection [Lee and Touba 2015]. However, in our application, MUX insertion is necessary to cause structural changes.

A six-input LUT can be instantiated as a 4:2 MUX, and inserted between an existing path of the design. The original path can pass through one of the inputs to the MUX, and the three other MUX inputs could connect to outputs from various LUTs that are candidates for fanout increase. However, during this process, a combinational loop should not be formed.

3.2.4 Obfuscation Process. Here we discuss how the transformation techniques described earlier could be used for obfuscating critical and non-critical segments.

Obfuscation of Non-critical Segments: As shown in Figure 8(c), a set of candidate non-critical LUTs and their corresponding features that could be modified to make them similar to critical LUTs is obtained from the uniqueness analysis. The candidate non-critical LUTs should meet the following criteria: (1) These LUTs are marked as true negative during uniqueness analysis, and (2) they have dark silicon available. Requirement (2) is relaxed if there are not enough candidate LUTs for applying a key insertion, and, hence, the ones that do not have dark silicon could be transformed with segmentation to create unused inputs. Table 1 shows how the different transformation techniques could be used for reducing the differentiating features among critical and non-critical functions. It is challenging to suggest how many non-critical (or critical) LUTs should be transformed. However, the increase in attack space and corresponding time to execute the attack can be estimated for a given number of false detection of critical and non-critical LUTs. The goal is to select transformations that lead to an impractical number of choices for the attacker. The misclassification can be estimated by performing uniqueness analysis after all transformations are applied. In Section 5.3, we demonstrate such an approximation of security for a given number of misclassification.

Generation of Obfuscated Variants of Critical Function: For the critical function, three different variants are generated each having different levels of imposed similarity to the non-critical. The algorithmic flow is presented in Algorithm 5 in Appendix A. As shown in the process, the critical LUTs that are identified correctly during the uniqueness analysis (i.e., C_{org}) are listed for modification to enforce similarity to the non-critical segment in various degrees. The degree of similarity is controlled by tuning different variables in P_t . For instance, during key insertion, for a given LUT in C_{org} , multiple unused inputs could be available. By randomizing the variables in P_t for each transformation, we can ensure that for each LUT, different numbers of key(s) are inserted with different orders of shuffling. The corresponding correct key values could be varied as well. Besides, the exact feature value attained after transformation could be varied within a bound. Once all suggested transformations are applied to all LUTs in C_{org} , a single variant is generated. This process is repeated for generating three variants. Due to the randomization of P_t , each LUT in a given variant and LUTs among the three variants become slightly different from each other.

Even though Algorithm 5 contains nested for-loops, with only three variants and six possible transformations, the required time for variant generation primarily depends on the total number of transformations required. For each transformation, there are two primary tasks: (1) randomization of P_t and (2) application of the transformation. Hence, from the variables used in Algorithm 5, the total time required to generate three variants can be estimated using the following equation:

$$\sum_{i=1}^3 \sum_{j=1}^{len(C_{org})} \sum_{k=1}^{len(T_j)} (Rd_{i,j,k} + Tx_{i,j,k}). \quad (3)$$

In the equation above $len(C_{org})$ denotes the total number of LUTs in C_{org} and $len(T_j)$ denotes the number of transformations needed for j th LUT. $Rd_{i,j,k}$ and $Tx_{i,j,k}$ denote the time required for randomizing the parameters in P_t and applying the k th transformation to the j th LUT that belongs to the i th variant.

Fig. 11. The figure shows the software flow of the proposed approach. Netlist from the original RTL is parsed using a Python script to extract feature data for each LUT. The data are used for uniqueness analysis from which candidate critical and non-critical LUTs are chosen for obfuscation with suitable transformation. The obfuscation and integration process is done using a Python script that outputs variant integrated netlist.

3.2.5 Integration of the Final Design. After variant generation, the three variants have to be integrated with OVUs and VMUs for each of their comparable outputs. All OVUs and VMUs are transformed as well such that they appear similar to the non-critical segments. The appropriate transformations for OVUs and VMUs could be obtained by using the same flow applied for critical functions. To find out the suitable internal signals for driving KAM ($F[0 : f]$), the design containing variants, OVUs, and VMUs is simulated for z cycles with $K[0 : k]$ applied temporarily from primary input in the absence of KAM. During each cycle, a desired set of wrong key values and an expected set of input vectors to the IP are applied. By observing all internal signals of the unobfuscated part of the design, a set of candidate internal signals are selected that are guaranteed to generate the same set of vectors until z clock cycles after start-up. A limited number of signals from the candidate nets are randomly selected to form $F[0 : f]$. These signals would be used for traversing KAM to the valid key application and monitoring states. Based on the varying wrong keys to apply, the correct key, the correct VMU responses, and the value of $F[0 : f]$ until z cycle, the KAM is designed in RTL. Before the integration of KAM to the obfuscated design, it could be separately synthesized to a netlist. After the integration of all modules, the final design is simulated for functional correctness, and the final bitstream is generated using a vendor-specific tool.

4 TOOLFLOW

In this section, we elaborate on the toolflow for generating the secure bitstream. As illustrated in Figure 11, it primarily contains an FPGA synthesis tool, two Python scripts for the netlist analysis and modification, and an open-source machine learning library. Below we describe their application.

4.1 Netlist Generation and Identification of Critical Function

Synplify Pro FPGA synthesis tool provides the technology netlist from the synthesized RTL for most FPGA vendors. Similar netlist could be obtained from the proprietary synthesis tools of the FPGA vendors as well. If the designer is concerned about an individual signal or a function that is not specified as a submodule, then it could be difficult to find the corresponding LUTs into the generated netlist. One way to overcome this problem is to instantiate a buffer using a single input LUT primitive and pass the critical signal or the signals of the target function through that buffer in the RTL code. The generated netlist does not modify the name of such LUTs and thus has been found to be useful to locate critical logic in the synthesized netlist. Once the inserted LUT is found, the preceding LUTs until the desired level could be protected. If the critical function is available as a separate submodule, then synthesis settings could be modified to provide a hierarchical netlist for easier identification of the critical module. Once the critical function is identified, the LUTs are labeled as either critical or non-critical for subsequent analysis.

4.2 Feature Data Extraction and Classification

For the purpose of uniqueness analysis, feature data have to be collected. We implemented a netlist parser written in Python to extract the desired functional and structural features for all LUTs within the design. The Python script outputs a .CSV file containing the feature data for each LUT labeled as critical or non-critical. As mentioned in Section 3, 10-fold cross-validation is executed on the extracted data to identify which critical and non-critical LUTs are easily distinguishable and require transformation. We used the machine learning library of Weka [Hall et al. 2009] to implement a Random Forest classifier. Selection of the appropriate classifier could depend on the data itself. Similar machine learning-based function identification has been done for hardware Trojan detection in soft-IPs [Hoque et al. 2018], where Random Forest has been found best for purely combinational Trojan function. Hence, for combinational S-Box identification, we used the same classifier. However, as demonstrated in Hoque et al. [2018], a voting ensemble of multiple classifiers (e.g., Random Forest, Naive Bayes, etc.) can be used to get optimum classification performance across all types of functions. After classification, a set of critical and non-critical LUTs is identified. Again, using a similar machine learning library, feature selection-based analysis is performed to rank the used features based on their effectiveness in correctly identifying the critical and non-critical LUTs. We used a wrapper method as the feature selection algorithm with the Random Forest classifier as the mining algorithm for the wrapper. From this analysis, the most influential features are found, which eventually helps to select suitable transformations for each LUT to maximize false detection.

4.3 Obfuscation and Integration

A Python script is used to apply the transformation to the original netlist. The required length of the obfuscation key is estimated and randomly generated by the script based on the required number of LUTs to be transformed. First, the non-critical LUTs are obfuscated followed by the generation of three variants of the critical function. The script also integrates the OVU and VMU for each critical function output. Once the variant, OVU, VMU integrated code is generated, simulation is done to obtain the input signals to KAM. As described earlier in Section 3.2, The KAM is generated by writing an RTL code based on the expected inputs, state transitions (valid/invalid key application states), and desired outputs (obfuscation key). This segment could also be automated, since the KAM architecture depends on a defined set of input parameters. The final KAM-integrated netlist is provided to a vendor-specific tool for generating the tamper-resistant bitstream.

5 RESULTS AND ANALYSIS

In this section, we demonstrate the effectiveness of our method in hiding a critical function by transforming critical and non-critical segments of the design. We also estimate the overhead for integrating multiple variants of the critical function in several cryptographic designs.

5.1 Uniqueness Analysis of S-Box

The goal of this analysis is to show the effectiveness of our approach in masking a critical function against an attacker with the ability to analyze the netlist using structural and functional features. For this purpose, we create a proof-of-concept System on Chip (SoC) design in RTL that contains an AES core, a DLX processor, and an FFT module. We synthesized the SoC for one of the Artix-7 FPGAs and obtained the netlist that uses LUTs and FFs only. However, to ensure that we find the S-box in the synthesized netlist, the hierarchy is preserved. Since our focus is to assess the obfuscation quality instead of attacking the AES, the use of hierarchical netlist should be acceptable.

To perform the uniqueness analysis, for each of the LUTs in SoC, we obtained the following features: (1) non-linearity of the contents, (2) maximum unexpected distance (MaxUD),

Fig. 12. Discernibility of AES S-Box before and after transformation: (a) Red data points represent 1,006 S-Box LUTs, and blue data points show 19,513 non-S-Box LUTs; (b) overlapping of red and blue data points after various four- and five-input non-S-Box LUTs transformed to non-linear six-input LUTs using key insertion; (c) data points of (a) with S-Box LUTs segmented; (d) data points from (c) after applying key insertion to non-critical LUTs targeting two S-Box clouds.

(3) maximum value of the Walsh spectrum (MaxWS), known as the function's weight, (4) entropy of the content, (5) input size of the LUT (fan-in), and (6) fanout of the LUT. As shown in Table 2, there is a total of 19,513 non-S-Box LUTs in SoC and 1,006 S-Box LUTs after synthesis. The size of the training data with 20,519 labeled observations, each with six features was 0.56MB. It took 1.57s to build the model on this dataset using a Random Forest classifier in the Weka tool. After 10-fold cross-validation, 935 S-Box LUTs were correctly identified with a small number of LUTs as false positives. The reason behind such an easy identification can be understood from the 3D plot of LUTs of the original design using function size, MaxUD, and MaxWS as shown in Figure 12(a). The availability of a large number of LUTs with fewer than six inputs can also be observed. Most of the 1,006 S-Box LUTs are visibly isolated from the rest of the 19,513 non-S-Box LUTs. In such a design, the S-Box can be separated only with a simple analysis of non-linearity related features of the LUT configuration. After applying the wrapper-based feature selection method, the non-linearity, MaxUD, function size, and fanout were found to be the most important features that help in archiving such high detectability in cross-validation.

To understand the impact of obfuscating non-critical LUTs (non-S-Box in this context), most of the available four- and five-input non-S-Box LUTs are first obfuscated with key insertion transformation. The key was inserted in such a way that the added contents caused the non-linearity value of the transformed LUT to fall within the known range of the S-Box function. Therefore, when the 10-fold cross-validation is repeated after non-critical modification, the number of false positives increased significantly (Table 2). The modification in non-linearity in non-S-Box LUTs caused an overlap between critical and non-critical segments. The corresponding plot is shown in Figure 12(b), with a significant overlap between critical and non-critical LUTs. Since all the impactful features observed from feature selection are not transformed yet, a lower number of false detection is expected even with such overlap in other features.

Table 2. Critical Function Classification Results for Various Transformations

Transformations Applied	Critical LUTs	Non-Critical LUTs	True Neg.	True Pos.	False Pos.	False Neg.
No Transformation	1006	19513	19429	935	84	71
Key Insertion in Non-Critical (KIN)	1006	19513	19161	911	352	95
+ Dummy fanout in Critical	1006	19513	19293	447	220	559
+ Segmentation of Critical & adjusted KIN	1246	19513	18934	412	579	834

We observed that most S-Box LUTs have an identical number of fanout. Therefore, we transformed the S-Box LUTs by adding dummy fanout in varying amounts. These fanout signals can be connected to various 2:1 or 4:2 MUXs that could be instantiated as separate LUTs. Here, these dummy fanout signals are connected to one of the unused inputs of non-critical LUTs where multiple unused inputs are available. The rest of the unused inputs are inserted with keys such that the dummy fanout signal has no control over the LUT output. Since the non-linearity between S-Box and non-S-Box LUTs are already overlapping, diversity in S-Box LUT fanout caused many of the S-Box LUTs to go undetected (Table 2). This indicates that even though the non-linearity related features were overlapping between critical and non-critical until now, the classifier was able to identify the S-Box LUTs using fanout feature, which is only available at the netlist-level.

Finally, to further achieve a higher number of false detections, we modified the S-Box LUT contents. Since the S-Box LUTs do not have any unused inputs, segmentation was applied for a limited number of LUTs. Two hundred forty LUTs among 1,006 were segmented, but no key was inserted. Segmentation alone may not be able to cause overlap between critical and non-critical LUTs unless they are appropriately transformed with key insertion. This can be observed in Figure 12(c), where a separate cloud of several five-input S-Box LUTs is observed in red. These five-input LUTs are the ones that connect to the inputs of the 2:1 MUX after segmentation. To ensure overlapping, the key insertion process of non-critical LUT has to be guided following two desired ranges of non-linearity values instead of one as was initially done in Figure 12(b). Ideally, features should be impacted such that they contain diverse values within LUTs of the same class, and the LUTs from opposite classes should overlap. Therefore, after segmentation, we apply key insertion to the non-critical LUTs such that both six- and five-input S-Box LUTs are covered by non-S-Box LUTs (Figure 12(d)). An increased number of false positives and false negatives are observed after the segmentation of critical LUTs and key insertion of non-critical LUTs. This number can further be improved if the majority of the S-Box is segmented and inserted with a key appropriately. This result should be consistent even when we integrate three variants of the critical function.

5.2 Overhead Analysis

In this section, we estimate the overhead of the proposed approach. Several cryptographic (e.g., AES, DES, SIMON) and large (e.g., b_18 contains several processors) benchmarks are chosen from ITC and Open Core suite [Basto 2000; OpenCores 2018]. Among those designs, specific modules are targeted for protection. The choice was based on the functionality of the modules. For example, for the crypto benchmarks, modules that function with the encryption key is chosen. For the b_18 benchmark, individual processor cores have been chosen to observe the overhead while protecting a large module. First, we used Synopsis Synplify Pro to get the synthesized netlist for each benchmark and provided that to our tool. For each critical module, we implemented the traditional TMR with no obfuscation as suggested in Trimberger [2007] and Drimer [2008] along with the proposed approach. We applied shuffling, key insertion, and fusion for obfuscation. Once the secure netlists are fully integrated with variants OVU, VMU, and KAM, synthesis, mapping, and PAR were done

Table 3. Comparison of Area Overhead between TMR and the Proposed Approach

Benchmark	Module	Original			TMR		Proposed		
		Total LUTs	Critical LUTs	% of Critical	Total LUTs	Inc. from Original (%)	Total LUTs	Inc. from Original (%)	Inc. from TMR (%)
b_18	80386 μP	9,531	1258	13.20	12,072	26.66	12,411	30.22	2.81
b_18	Viper μP	9,531	907	9.52	11,418	19.80	11,571	21.40	1.34
AES	AddRoundKey	967	206	21.30	1,781	84.18	1,822	88.42	2.30
AES	controller	967	63	6.51	1,016	5.07	1,040	7.55	2.36
DES	Key_gen	378	57	15.08	550	45.50	550	45.50	0.00
FPU	Add_sub	3,381	29	0.86	3,485	3.08	3,476	2.81	-0.26
USB_FUNC	CRC16	1,938	12	0.62	1,983	2.32	1,984	2.37	0.05
Simon	key_exp.	54	34	62.96	129	138.89	143	164.81	10.85
Avg.		3343.38	320.75	16.26	4054.25	40.69	4124.63	45.39	2.43

using Xilinx ISE Webpack 14.7 tool. Xilinx Spartan6 FPGA has been used for most benchmarks. For the largest b_18 benchmark, we used one of Xilinx Artix7.

5.2.1 Area, Power, and Delay Overhead. Table 3 summarizes the area overhead results. For each benchmark, the areas of both the original design and the corresponding target module are reported in terms of LUTs. The protected areas are 16% of the whole design on average. After implementing TMR, an average overhead of 40% is observed. Ideally, no additional hardware is needed for shuffling, key insertion, and fusion. The overhead of the proposed approach compared to TMR primarily stems from the integration of VMU and KAM. Overhead due to VMU is dependent on the number of outputs of the critical function. The size of KAM increases with the size of the key being generated and VMU responses to be observed. The key size from KAM depends on the size of the critical function. For larger critical functions a key size of 128 bit is generated and shared among the obfuscate LUTs. Multiple KAMs could be used instead of just one to generate the obfuscation key in segments, which would reduce the fanout of each KAM. Finally, we observe that the proposed approach incurs only 2.43% more area on average compared to traditional TMR while providing a significantly higher resistance against tampering. The security benefits of the proposed approach compared to TMR will be discussed in Section 5.3. Area overhead will increase if segmentation is incorporated in the absence of dark silicon.

Table 4 summarizes the power overhead results. Similar to the area overhead, a slight increase in power has been observed for most benchmarks when compared with TMR. To protect 16% area (from Table 4), an average of 29.4% power penalty compared to the original design is observed. As seen in Figure 13(a), the corresponding power overhead would also be admissible as long as the ratio of the protected area to the whole design is small. The delay results are organized in Table 5. Since only the OVUs are added inside each path, the resultant delay overhead is low. The delay mostly stays the same as the original design after applying three different transformations. An average delay overhead of 1.08% is observed compared to the original design.

Figure 13(a) illustrates the change of overhead for implementing both traditional TMR and the proposed approach. The blue dotted line is the percentage of area being protected for each module. It is clear that the resultant overhead is equivalent to the percentage of the area we are protecting against tampering. To monitor a smaller module within a large design, the overhead would be very low. Figure 13(b) shows the change in total energy calculated from Tables 4 and 5. The energy is calculated as nanojoule by multiplying the total power (Watt) and delay (ns) for each design under

Table 4. Comparison of Power Overhead between TMR and the Proposed Approach

Benchmark	Module	Original			TMR			Proposed				
		Total (mW)	Static (mW)	Dynamic (mW)	Total (mW)	Static (mW)	Dynamic (mW)	Total (mW)	Static (mW)	Dynamic (mW)	Inc.from Original (%)	Inc. from TMR (%)
b_18	80386 μP	267	117	150	305	117	188	347	117	230	29.9	13.7
b_18	Viper μP	267	117	150	302	117	185	284	117	167	6.3	-5.9
AES	AddRoundKey	97	65	32	131	65	65	180	66	113	85.5	37.4
AES	controller	97	65	32	104	65	39	147	66	81	51.5	41.3
DES	Key_gen	251	32	219	323	34	289	332	34	299	32.2	2.7
FPU	Add_sub	79	64	15	80	64	16	80	64	16	1.2	0
USB_FUNC	CRC16	286	82	204	321	83	239	323	83	241	12.9	0.62
Simon	key_exp.	88	82	6	102	82	20	102	82	20	15.9	0
Avg.		179	78	101	208.5	78.3	130.125	224.3	78.6	145.8	29.4	11.2

Fig. 13. The figure shows the change in overhead to incorporate TMR and variant-based approach (VM) for different sizes of critical functions: (a) area overhead increases with the percentage of the protected area for both techniques; (b) energy overhead reduces when a small area is covered within a large design.

original, TMR, and proposed approach. From there, the percentage of the increase in energy due to both TMR and the proposed approach is found, which is plotted in the figure as a column chart. The green line shows the size of the original design for corresponding benchmarks and the blue line shows the percentage of the area being protected. These two lines help to observe that if the original design is large and the corresponding protected area is small, the resultant energy overhead would be small. Therefore, security analysis to identify the smallest possible area under attack could help the designer to reduce the area and power overhead.

5.2.2 Overhead Analysis of Individual Transformations. To compare the area, power, and delay overhead of five different transformations discussed in this article, we implemented them on a set of ISCAS89 benchmarks using Artix-7 FPGA (xc7a100tcs324). For each design, the five transformations were applied separately to generate one variant per transformation. For each transformation applied to a design, 50 LUTs were targeted to ensure coherency. However, for fusion, a very few LUT pair candidates were available for applying the transformation. Hence, for the rest of the LUTs, we used key insertion. Unlike the earlier overhead analysis, multiple variants, OVU, VMU, or KAM were not integrated to facilitate a more precise comparison of the overhead only due to different transformations. As shown in Figure 14, compared to all other transformations, segmentation causes the highest increase in the required number of LUTs. For smaller designs,

Table 5. Comparison of Delay Overhead between TMR and the Proposed Approach

Benchmark	Module	Original Period (ns)	TMR Period (ns)	Proposed	
				Period (ns)	% Inc. from Original
b_18	80386 μP	12.2	12	12	-1.63
b_18	Viper μP	12.2	13.7	12	-1.63
AES	AddRoundKey	7.7	8	8	3.89
AES	controller	7.7	7.7	7.7	0
DES	Key_gen	6.2	6.2	6.8	9.67
FPU	Add_sub	112	112	112.5	0.44
USB_FUNC	CRC16	1.9	1.9	1.8	-5.26
Simon	key_exp.	2.2	2.2	2.27	3.18
Avg.		20.2	20.4	20.3	1.08

Fig. 14. The figure shows the (a) area, (b) dynamic power, and (c) delay analysis for five transformations across different benchmarks.

this translated to a larger impact on dynamic power and delay. Since shuffling did not incur any change to any parameter, it is shown with the original results. Due to the modification of path length, MUX insertion and segmentation caused a greater increase in delay. This impact could be reduced if the critical path information is considered during the transformation assignment.

5.3 Security Analysis

The threat models presented in Section 2.1 defined three bitstream alteration attacks: TM, RM, and RBM. While TM applies to unencrypted bitstream only, RM and RBM are applicable to both encrypted and unencrypted configuration files. Below we consider the two worst-case scenarios where the attacker can compromise a critical function successfully with the smallest possible simultaneous alteration of LUTs. We assume that simply inverting each bit of LUTs would be sufficient. In practice, a more complex alteration could be required for a successful attack.

- **Case 1:** For each output of the critical function, LUTs in (at least) two corresponding variants have to be altered simultaneously (Figure 15(a)).
- **Case 2:** The OVU-VMU pair has to be modified for each output signal of the critical function (Figure 15(b)).

Below we discuss the number of attempts needed to compromise the system under these two cases.

Fig. 15. Panels (a) and (b) show Case 1 and Case 2 attack scenarios, respectively, that require minimum alterations. We analyze the security based on these two worst-case scenarios.

5.3.1 Security against TM. While performing TM, the attacker's goal is to reduce the attack space through static analysis. As we have shown in Section 5.1, a target logic with known identifying features can be separated from a large design with several thousands of LUTs. However, we have also shown that when these features are impacted through targeted transformations, a large number of false detection occurs. While an increased false positive (i.e., non-critical LUTs identified as critical LUTs) expands the search space for the attacker, the presence of false negatives (i.e., critical LUTs identified as non-critical LUTs) can completely thwart such attacks. Let us assume a critical function with O_{CL} different outputs has N_V different LUTs in each variant (assumed equal for simplification) with N_{OVU} different OVUs and N_{VMU} different VMUs. We assume that the LUT that drives the output of a variant to its OVU is the best target for an attacker among all other LUTs in that variant. For instance, in Figure 15(a), the signal $C1_{out}$ within the cloud can be inverted by altering the final driving LUT in fan-in cone marked in red. Hence, in the worst-case, $N_V=1$ for inverting an output signal of a variant. To execute Case 1, the attacker has to simultaneously compromise two such variants to defeat our technique in this worst-case example. If the identification process of variants incurs false negatives, then the attack will no longer work. In the presence of false positives (FP_V) only, attacker's search space (SP) to correctly implement Case 1 would become

$$SP_{Case1} = \binom{2 \times N_V + FP_V}{2 \times N_V}. \quad (4)$$

N_V is multiplied by two as at least two of the variants must be altered. In practice, the driving LUTs of different output signals that belong to the critical function could be different in nature, requiring separate learning and identification process. In such a case, SP_{Case1} would contain multiplication of O_{CL} different binomial coefficients instead of just one. Even if the driving LUTs of all the output signals of a critical function is similar, the required number of attempts would still be high. To understand the presented equation for Case 1, let us discuss using a simple example. A critical function with only two output signals driven by two LUTs with similar features ($N_V = 2$) is transformed to generate variants. If FP_V of 200 is achieved due to transformation, then the estimated number of trials to execute Case 1 would be $\binom{2 \times 2 + 200}{2 \times 2} = 70,058,751$. The shortest reported time to alter a bitstream, reprogram, and observe the outcome is 3.3s [Swierczynski et al. 2018]. However, this duration could vary depending on the clock speed and the size of the design under attack. According to that, the required time to test all possibilities would be 2675.8 days. Transformations could be added such that the number of false positives would be significantly higher compared to the above example. If the two output signals were different in nature, then the required trials would become $\binom{2 \times 1 + 200}{2 \times 1} \times \binom{2 \times 1 + 200}{2 \times 1} = 412,130,601$, which requires 15,741 days to execute.

Fig. 16. Panels (a) and (b) provide an estimation of the number of days required to compromise the proposed technique with TM attack under Case 1 and Case 2 scenario, respectively. With increased number of false positives the attack difficulty becomes comparable to compromising encrypted designs with RBM. Figure (c) evaluates Case 1 and Case 2 scenario against RBM attack and compares their security with encrypted and unencrypted designs.

Similarly to Case 1, Case 2 search space that requires concurrent alteration of OVU and VMU can be estimated as

$$SP_{Case2} = SP_{OVU} \times SP_{VMU} = \binom{N_{OVU} + FP_{OVU}}{N_{OVU}} \times \binom{N_{VMU} + FP_{VMU}}{N_{VMU}}. \quad (5)$$

In the equation above, N_{OVU} is the number of output signals coming from different OVUs. FP_{OVU} denotes their corresponding false positives. The same applies for N_{VMU} and FP_{VMU} . Using the aforementioned equations, we estimate the number of days required to compromise our system under Case 1 and Case 2 scenario for various functions. Assuming $N_V = 1 - 4$, $FP_V = 100 - 400$, and $T_{trial} = 3.3s$, the required number of days to try all possibilities to execute Case 1 is presented in Figure 16(a). Since encrypted designs cannot be impacted by TM, we compare our method with an encrypted design's security against RBM attack (taken from the subsequent RBM analysis). Figure 16(b) shows a similar plot for Case 2 assuming $N_V = 1 - 4$, FP_{OVU} and $FP_{VMU} = 100 - 400$. Unlike Case 1, Case 2 always contains a multiplication of two different binomial coefficients. Therefore, if a high FP_{OVU} and FP_{VMU} (i.e., >200 for both) can be achieved, the effort required for compromising through Case 2 becomes higher than mounting RBM in an encrypted design. Finally, if we consider the presence of false negatives for any of the cases, then the attacker would not be able to succeed even with a large number of attempts.

5.3.2 Security against RM. To perform an RM attack, the adversary randomly alters the configuration file without any knowledge of its format. Unlike TM, no static analysis is performed to reduce the search space. However, this attack is also applicable to the encrypted bitstream. If the critical function in the original design only depends on one LUT, then even an encrypted version of the design could get accidentally compromised. If a total of B_{cnf} configuration bits constitute the bitstream, then alteration of one bit at a time would require B_{cnf} different trials. If the bitstream size is small (e.g., few Kilo Bits) or unencrypted header information is eliminated by the attacker, then the attack vector becomes very practical. Encrypted/unencrypted bitstream files used for partial configuration are typically small. For instance, if the reconfigurable partition uses 20% of the FPGA resources, then the corresponding partial configuration file will be almost 20% of the full design bitstream size. The proposed countermeasure increases the search space for both Case 1 and 2 significantly. For a simple one output critical function, both cases require simultaneous alteration of at least two LUTs. If the attacker only flips two bits in parallel, then all possible trials would be $B_{cnf} \times (B_{cnf} - 1)$ for compromising the single output function. Conversely, for an encrypted or unencrypted design, RM would only require B_{cnf} different trials in the absence of the proposed solution.

5.3.3 Security against RBM. Rule-based modifications are more advanced attack models compared to RM. RBM follows a fixed set of modification rules (e.g., clear/set/invert whole/half LUT/slice) instead of modifying a single bit randomly. Certain modifications can even be applied to an encrypted bitstream. For instance, an attacker can invert individual LUTs even in the encrypted configuration file by obtaining a deep understanding of the format. However, in the encrypted bitstream, the completely unused LUTs cannot be identified, which forces the attacker to modify all LUTs of the FPGA (N_{FPGA}).

Since an attacker can invert all bits of a single LUT at a time, to compromise a critical function with O_{CL} outputs in unencrypted design containing N_{dsq} LUTs, a total of $\binom{N_{dsq}}{O_{CL}}$ attempts are required. Hence, if $O_{CL} = 2$ and $N_{dsq} = 1000$, then an unencrypted design without the proposed countermeasure can be compromised with $\binom{1000}{2} = 499,500$ attempts. However, when the proposed countermeasure is in place, under both Case 1 and 2, for compromising each output in O_{CL} , at least two LUTs have to be altered simultaneously. Hence, $\binom{1000}{2 \times 2} = 41,417,124,750$ attempts are required. Assuming $N_{CL} = 1 - 4$, $N_{dsq} = 1000$, $N_{FPGA} = 30,000$, and $T_{trial} = 3.3s$, the required number of days to try all possibilities are presented in Figure 16(c). It is clear that our technique provides better security compared to stand-alone encryption against rule-based modifications.

5.3.4 Comparison of Security with TMR. We also compare our security against TM and RBM with TMR approach for which we observed the overhead. Since TMR does not come with any obfuscation, all copies of the critical function should be easily observable during TM attack. However, due to placement and routing, a limited amount of variance among each copy of the function is possible. Introducing constraint-induced variation among the redundant copies has previously been suggested for tolerating Trojan in FPGA silicon [Mal-Sarkar et al. 2016]. However, this does not guarantee security against TM, since in Swierczynski et al. [2015a], S-Box was located in eight different implementations of the AES (with varied architecture and implementations) utilizing the same non-linearity feature. Such signatures inherent to the Boolean logic of the critical function is unlikely to be eliminated with varying implementation constraints. However, for our approach, such inherent signatures are systematically eliminated, causing a large number of false positives and false negatives during static analysis. TMR also does not provide any additional benefit against the RBM attack unless the voting logic is monitored/protected separately. Through RBM, the attacker can observe the desired malicious outcome by altering only the voting logic of TMR. Hence, the security of altering each critical signal reduces to altering the corresponding LUT implementing the voting logic. Hence, the quantitative security of TMR against RBM is equivalent to the security of “no solution” as reported in Figure 16(c).

5.4 Additional Attacks

Here we discuss some of the potential structural and dynamic analysis-based attacks that are only feasible if the netlist can be obtained through reverse engineering of the bitstream. The attacker would also have to obtain the inputs that are typically applied to the FPGA mapped IP after system startup to observe a normal operation in simulation.

5.4.1 Identification of KAM. Identification of KAM may not break the proposed system directly but could provide useful information to mount more complicated attacks. Since the KAM is responsible for generating the obfuscation keys, it is expected to have a high fanout. However, this could be avoided by instantiating multiple KAMs, each of which provides a part of the obfuscation key.

Another way to identify the state elements of KAM would be to perform dynamic analysis to isolate FFs that do not switch during normal operation of the system. However, we propose to incorporate multiple states for the key application and monitoring phase where KAM would

traverse continuously after certain clock cycles. This would create activity within the KAM circuitry throughout any simulation of the design.

5.4.2 Identification of Key Inputs. A dynamic analysis could be performed on the netlist after the design becomes functional to understand which nets do not change during normal operation. Since key inputs are always asserted, it could be possible that the attacker identifies such stable nets as key inputs. This is a challenge even for any key-based obfuscation technique applicable to FPGA designs. This could be evaded if the variants could be operated with time-varying keys, all of which provide correct functionality. For example, for a given correct obfuscation key, a variant-pair could produce identical outputs, and for another correct obfuscation key, their outputs could be always the opposite. Based on one of the key inputs to OVU and VMU such a change in variant-pair functionality can be integrated. Therefore, observation of static signals would no longer be useful.

5.4.3 Identification of OVU-VMU Pairs. Since the VMU for given OVU shares three common inputs, and the VMU takes the output of OVU, an attacker could try to structurally analyze the design to find such LUT pairs. However, such analysis would provide high false positives as they are not uncommon in designs. Moreover, such a structure can be eliminated by segmenting the VMU. After segmentation, one of the variant inputs or OVU output could go to the MUX connecting the segmented VMU LUTs. Hence, the known structure would no longer be present. Besides, if LUT pairs that are suitable for fusion have many unused inputs, they could be transformed with three common key inputs to appear like OVU-VMU pair.

6 CONCLUSION

We have presented an effective and scalable solution to achieve secure field operation of FPGA in the presence of bitstream tampering attacks without support from cryptographic solutions. It introduces an obfuscation-based countermeasure and corresponding toolflow that is suitable for protecting critical regions of design in FPGA. The toolflow is amenable for seamless integration with the commercial toolchain. Unlike traditional approaches, e.g., TMR, the proposed tamper-tolerance approach can achieve a high level of security against targeted modifications. Moreover, against rule-based modification, our technique is more effective compared to encryption-based solutions. We also have shown that by leveraging the unused memory inside the LUTs, we can increase the complexity of the obfuscation while keeping the overhead low.

APPENDICES

A ALGORITHMS FOR APPLYING TRANSFORMATIONS

ALGORITHM 1: Generating shuffled content for a single LUT

input: Desired shuffled sequence S (i.e., [3 1 2 4]), I_{org} : Input side of the truth table as 2-D array where each row represents one input sequence, O_{org} : Array containing output for each input sequence
output: Shuffled LUT content represented as O_{new}
 // First, create I_{new} by shuffling the input columns of I_{org}
 I_{new} = New 2-D array with size of I_{org}
foreach column index c in I_{org} **do**
 | $I_{new}[S[c]] \leftarrow I_{org}[c]$
end
foreach row index $r2$ in I_{new} **do** // Now create the shuffled output column
 | **foreach** row index $r1$ in I_{org} **do**
 | | **if** ($I_{new}[r2] == I_{org}[r1]$) **then**
 | | | $O_{new}[r2] \leftarrow O_{org}[r1]$
 | | **end**
 | **end**
end // O_{new} contains the shuffled LUT content

ALGORITHM 2: Generating key inserted content for a single LUT

input: O_{org} : Array containing original LUT contents, Z : Current input size, K : Value of the key for correct operation
output: O_{new} : LUT content after key insertion
 New input size = $Z + 1$
 O_{new} = New array of size $2^{(Z+1)}$
if ($K == 0$) **then**
 | $O_{new}[1 : 2^Z] \leftarrow O_{org}$
 | $O_{new}[2^Z : 2^{(Z+1)}] \leftarrow \text{Random}$
end
else
 | $O_{new}[1 : 2^Z] \leftarrow \text{Random}$
 | $O_{new}[2^Z : 2^{(Z+1)}] \leftarrow O_{org}$
end // Shuffling is done after key insertion to move the key input.

ALGORITHM 3: Generating contents after fusion for a LUT pair

input: Two consecutively connected LUTs having at least one unused input. O_{orgR} : rear LUT content. Z_R : rare LUT input size, O_{orgF} : front LUT content, Z_F : front LUT input size, K : correct key value

output: modified LUT contents after fusion, represented as O_{newR} and O_{newF}

New input size = $Z_R + 1$

O_{newR} = New array of size $2^{(Z_R+1)}$

if ($K == 0$) **then** // Apply key insertion in rear LUT

- | $O_{newR}[1 : 2^{Z_R}] \leftarrow \overline{(O_{orgR})}$
- | $O_{newR}[2^{Z_R} : 2^{(Z_R+1)}] \leftarrow \text{Random}$

end

else

- | $O_{newR}[1 : 2^{Z_R}] \leftarrow \text{Random}$
- | $O_{newR}[2^{Z_R} : 2^{(Z_R+1)}] \leftarrow \overline{(O_{orgR})}$

end

end procedure // Swap the contents of front LUT to accommodate the inversion of rare LUT

O_{temp} = new array of size 2^{Z_F}

$P \leftarrow$ position of the rear LUT's output among the inputs of front LUT Content length of front LUT,

$S_F = 2^{Z_F}$

for each i **in range** ($1 : S_F/2$) **do**

- | $O_{temp}[i] \leftarrow O_{orgF}[i + P]$
- | $O_{temp}[i + P] \leftarrow O_{orgF}[i]$

end

// Now generate O_{newF} from O_{temp} by inserting key

O_{newF} = new array of size $2^{(Z_F+1)}$

if ($K == 0$) **then**

- | $O_{newF}[1 : 2^{Z_F}] \leftarrow (O_{temp})$
- | $O_{newF}[2^{Z_F} : 2^{(Z_F+1)}] \leftarrow \text{Random}$

end

else

- | $O_{newF}[1 : 2^{Z_F}] \leftarrow \text{Random}$
- | $O_{newF}[2^{Z_F} : 2^{(Z_F+1)}] \leftarrow (O_{temp})$

end

ALGORITHM 4: Generating LUT contents after segmentation

input: O_{org} : original LUT content. Z : original input size

output: O_{S1}, O_{S2} : contents of segmented LUTs. O_C : Contents of the LUT combining O_{S1}, O_{S2}

L_{S1} and L_{S2} share common inputs $I_0 : I_{Z-1}$ L_C takes outputs from L_{S1} and L_{S2} , and the left out original input I_Z . L_C chooses between L_{S1} and L_{S2} depending on I_Z .

for each i **in range** ($1 : 2^Z$) **do**

- | $O_{S1} \leftarrow O_{org}[1 : (2^Z)/2]$
- | $O_{S2} \leftarrow O_{org}[(2^Z)/2 : (2^Z)]$

end

$O_C \leftarrow$ Contents of 2:1 multiplexer that selects between L_{S1} and L_{S2} based on I_Z

// Once L_{S1}, L_{S2} , and L_C are created, shuffling and key insertion are applied on each

ALGORITHM 5: Variant generation process

input: C_{org} : list of candidate critical LUTs. T : list of selected transformation(s) for each LUT in C_{org} . P_t : adjustable parameters for transformation // Example P_t variables: degree of change in LUT configuration, no. of key inputs inserted, input shuffle order and so on.

output: $C1, C2, C3$: three variants of critical function

Estimate applicable key size, L_k // estimate from required transformations (no. of key insertion, fusion, segmentation, etc., in T)

if ($L_k < Key_{min}$) **then**
 // minimum key size Key_{min}
 Alter T to include additional segmentation transformation to increase the key space

end

for each variant i among 1 : 3 **do**
 Generate Key for variant i with size L_k // different Key for $C1/C2/C3$
 for each LUT j in C_{org} **do**
 for each transformation k in T selected for j **do**
 Randomize P_t // Randomization of transformation parameters
 Apply transformation k to LUT j
 end
 end
 Integrate all transformed LUTs to form $C1/C2/C3$

end

ACKNOWLEDGMENT

We appreciate the valuable input from Dr.-Ing. Marc Fyrbiak regarding the Walsh Coefficient calculation.

REFERENCES

- Alaaddin Al-Omary, Haider M. AlSabbagh, and Hussain Al-Rizzo. 2018. Survey of hardware-based security support for IoT/CPS systems. *KnE Eng.* 3, 7 (2018), 52–70.
- Christoph Albrecht. 2005. IWLS 2005 benchmarks. In *Proceedings of the International Workshop for Logic Synthesis*.
- Luis Basto. 2000. First results of ITC'99 benchmark circuits. *IEEE Des. Test Comput.* 17, 3 (2000), 54–59.
- Prabuddha Chakraborty, Jonathan Cruz, and Swarup Bhunia. 2018. SAIL: Machine learning guided structural analysis attack on hardware obfuscation. In *Proceedings of the 2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST'18)*. IEEE, 56–61.
- Prabuddha Chakraborty, Jonathan Cruz, and Swarup Bhunia. 2019. SURF: Joint structural functional attack on logic locking. In *Proceedings of the 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST'19)*. 181–190.
- Rajat Subhra Chakraborty and Swarup Bhunia. 2009. HARPOON: An obfuscation-based SoC design methodology for hardware protection. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 28, 10 (2009), 1493–1502.
- Rajat Subhra Chakraborty, Indrasish Saha, Ayan Palchoudhuri, and Gowtham Kumar Naik. 2013. Hardware Trojan insertion by direct modification of FPGA configuration bitstream. *IEEE Des. Test* 30, 2 (2013), 45–54.
- Saar Drimer. 2008. Volatile FPGA design security—a survey. *IEEE Computer Society Annual Volume* (2008), 292–297.
- Mojtaba Ebrahimi, Parthasarathy Murali B. Rao, Razi Seyyedi, and Mehdi B. Tahoori. 2016. Low-cost multiple bit upset correction in SRAM-based FPGA configuration frames. *IEEE Trans. VLSI Syst.* 24, 3 (2016), 932–943.
- Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: An update. *ACM SIGKDD Explor. Newslett.* 11, 1 (2009), 10–18.
- Max Hoffmann and Christof Paar. 2018. Stealthy opaque predicates in hardware-obfuscating constant expressions at negligible overhead. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018, 2 (2018), 277–297.
- Tamzidul Hoque, Jonathan Cruz, Prabuddha Chakraborty, and Swarup Bhunia. 2018. Hardware IP trust validation: Learn (the untrustworthy), and verify. In *Proceedings of the International Test Conference*. IEEE, 1–10.

- Robert Karam, Tamzidul Hoque, Sandip Ray, Mark Tehranipoor, and Swarup Bhunia. 2016. Robust bitstream protection in FPGA-based systems through low-overhead obfuscation. In *Proceedings of the International Conference on ReConfigurable Computing and FPGAs*. IEEE, 1–8.
- Robert Karam, Tamzidul Hoque, Sandip Ray, Mark Tehranipoor, and Swarup Bhunia. 2017. MUTARCH: Architectural diversity for FPGA device and IP security. In *Proceedings of the 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC'17)*. IEEE, 611–616.
- Yu-Wei Lee and Nur A. Touba. 2015. Improving logic obfuscation via logic cone analysis. In *Proceedings of the Latin-American Test Symposium*. IEEE, 1–6.
- Huan Liu and Lei Yu. 2005. Toward integrating feature selection algorithms for classification and clustering. *IEEE Trans. Knowl. Data Eng.* 17, 4 (2005), 491–502.
- Heiko Lohrke, Shahin Tajik, Thilo Krachenfels, Christian Boit, and Jean-Pierre Seifert. 2018. Key extraction using thermal laser stimulation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018, 3 (2018), 573–595.
- Sanchita Mal-Sarkar, Robert Karam, Seetharam Narasimhan, Anandaroop Ghosh, Aswin Krishna, and Swarup Bhunia. 2016. Design and validation for FPGA trust under hardware Trojan attacks. *IEEE Trans. Multi-Scale Comput. Syst.* 2, 3 (2016), 186–198.
- Amir Moradi, David Oswald, Christof Paar, and Pawel Swierczynski. 2013. Side-channel attacks on the bitstream encryption mechanism of Altera Stratix II: Facilitating black-box analysis using software reverse-engineering. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 91–100.
- OpenCores. 2018. The Reference Community for Free and Open Source Gateway IP Cores. Retrieved from <https://opencores.org/>.
- Khoa Dang Pham, Edson Horta, and Dirk Koch. 2017. BITMAN: A tool and API for FPGA bitstream manipulations. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 894–897.
- Synplify Pro. 2018. Logic Synthesis for FPGA Design. Retrieved from <https://www.synopsys.com/implementation-and-signoff/fpga-based-design/synplify-pro.html>.
- Sergei P. Skorobogatov and Ross J. Anderson. 2002. Optical fault induction attacks. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2–12.
- Pedro Miguel Sosa. 2016. *Calculating Nonlinearity of Boolean Functions with Walsh-Hadamard Transform*. UCSB, Santa Barbara.
- Pawel Swierczynski. 2018. *Bitstream-Based Attacks Against Reconfigurable Hardware*. Ph.D. Dissertation. Ruhr University Bochum, Germany.
- Pawel Swierczynski, Georg T. Becker, Amir Moradi, and Christof Paar. 2018. Bitstream fault injections (BiFI)—Automated fault attacks against SRAM-based FPGAs. *IEEE Trans. Comput.* 67, 3 (2018), 348–360.
- Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, and Christof Paar. 2015a. FPGA Trojans through detecting and weakening of cryptographic primitives. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 34, 8 (2015), 1236–1249.
- Pawel Swierczynski, Marc Fyrbiak, Christof Paar, Christophe Hurioux, and Russell Tessier. 2015b. Protecting against cryptographic trojans in FPGAs. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 151–154.
- Shahin Tajik, Heiko Lohrke, Jean-Pierre Seifert, and Christian Boit. 2017. On the power of optical contactless probing: Attacking bitstream encryption of FPGAs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1661–1674.
- IoT Times. 2018. Why Software Cannot Effectively Secure Hardware. Retrieved from <https://iot.eetimes.com/why-software-cannot-effectively-secure-hardware/>.
- Steve Trimberger. 2007. Trusted design in FPGAs. In *Proceedings of the Design Automation Conference*. ACM, 5–8.
- Stephen M. Trimberger and Jason J. Moore. 2014. FPGA security: Motivations, features, and applications. *Proc. IEEE* 102, 8 (2014), 1248–1265.
- Xilinx. 2014. Vivado Design Suite User Guide: Synthesis, UG901 (v2014.1). Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug901-vivado-synthesis.pdf.
- Xilinx. 2015. Security Monitor IP Core. Retrieved from <https://www.xilinx.com/support/documentation/product-briefs/security-monitor-ip-core-product-brief.pdf>.
- Xilinx. 2019. Applications: Vehicle Networking and Connectivity. Retrieved from <https://www.xilinx.com/applications/automotive/vehicle-networking.html#spartan6>.

Received December 2018; revised July 2019; accepted September 2019