SFLL-HLS: Stripped-Functionality Logic Locking Meets High-Level Synthesis

(Invited Paper)

Muhammad Yasin, Chongzhi Zhao, and Jeyavijayan (JV) Rajendran
Department of Electrical and Computer Engineering, Texas A&M University
{myasin, cz_zhao, jv.rajendran}@tamu.edu

***Abstract*—Logic locking has emerged as a promising countermeasure against piracy and reverse engineering attacks on integrated circuits. The state-of-the-art logic locking techniques, more specifically stripped-functionality logic locking (SFLL), offer provable security guarantees against many attacks. However, these techniques focus on protecting individual modules or even parts of a module, failing to deliver system-wide security. This paper sheds light on integrating logic locking with high-level synthesis (HLS) in an attempt to deliver system-wide security. We demonstrate the integration of SFLL with LegUp HLS tool for an image processing application.**

I. Introduction

A. IP piracy and reverse engineering

The ever-increasing complexity of integrated circuits and the cost of establishing foundries have given rise to fabless companies [1]. Today, many companies, e.g., Apple and Qualcomm, outsource integrated circuit (IC) fabrication to offshore foundries, e.g., TSMC and Samsung, that are located mostly in Asia. While this outsourcing has helped subsidize IC manufacturing costs, it has also brought with it several security challenges in the form of IP piracy, overbuilding, reverse engineering, and hardware Trojans [2].

A number of design-for-trust (DfTr) countermeasures such as IC metering [3], watermarking [4], IC camouflaging [5], split manufacturing [6], and logic locking [7], [8], [9] have been proposed to tackle the aforementioned security challenges [10]. Logic locking surpasses the other DfTr techniques by offering protection against a wider range of threats, originating from either untrusted foundries or untrusted end-users, as illustrated in Table I. Logic locking is carried out earlier in the design flow at either register transfer level (RTL) or gate level. Split manufacturing and camouflaging, however, are conducted on the layout.

TABLE I
A comparison of DfTr techniques based on the abstraction level and the protection offered.

Technique	Abstraction level	Protection
Camouflaging [5]	Layout	End-user
Split manufacturing [6]	Layout	Foundry
Logic locking [7], [8], [9], [19], [14]	RTL/gate	Foundry + end-user
TAO [20]	RTL	Foundry
SFLL-HLS (this paper)	RTL	Foundry + end-user

B. Logic locking

Logic locking protects a circuit by introducing additional key-controlled logic into a circuit. In addition to the original inputs, a locked circuit has *key inputs* that are driven by an on-chip tamper-proof memory. The *locked* netlist passes through the untrusted design phases, i.e., untrusted manufacturing, test, and assembly. Without the knowledge of the secret key, neither a design can be pirated nor an IC can be made functional.

The earliest logic locking techniques that focus on inserting XOR/XNOR key-gates [7], [8] are vulnerable to attacks such as the SAT that leverages Boolean satisfiability to eliminate incorrect key [11]. Subsequent research on logic locking defends against the SAT attack by combining an original circuit with a point-function [9], [12]. However, the point function can be isolated using removal attacks [13]. An approximate key for these techniques can also be recovered by approximate attacks [14]. Stripped functionality logic locking (SFLL) is the first technique to defend against all these attacks in a provable way [15], [16]. Sequential locking techniques that lock finite state machines [3], [17] are susceptible to state-machine-reconstruction attacks [18].

We would like to emphasize that the focus of logic locking research has so far been on protecting individual modules or even combinational logic that is only a small part of a single module. Accordingly, the metrics and algorithms used in logic locking research are also not yet adequately suited for offering system-wide protection.

C. High-level synthesis (HLS)

Nowadays, high-level synthesis (see Section II-B for details) is being actively deployed successfully to design large scale SoCs. By shifting the design effort to a higher level of abstraction, HLS brings forth unique advantages in terms of faster validation of large designs, smaller time-to-market, and agile engineering change orders. HLS tools often support a variety of hardware configurations allowing the use of several types of functional units, storage elements, and on-chip interconnect.

We envision that logic locking can leverage the developments in HLS to effect system-level protection for large scale SoCs. As explained in Section II-B, HLS involves compiling the high-level code to an intermediate representation (IR). The HLS tools conduct scheduling and binding operations on this IR. The IR contains useful information about the control and datapath of a design. This representation can be used to identify suitable locations to be locked such that the protection for the overall system is enhanced. This will enable system-wide security and cost trade-offs. The cost can be area, power, latency, throughput, or any such metric of interest. The security metric(s) can be selected based on the threat model. Another potential advantage of HLS is that it can help identify the

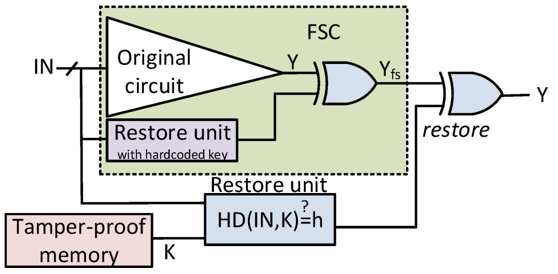


Fig. 1. The architecture of SFLL-HD. The overall circuit consisting of the original circuit, the restore unit, and a restore unit with the secret key hard-coded is synthesized using logic synthesis tools [15].

modules that are most crucial from security stand-point early in the design process.

TAO is the only research work considering both logic locking and HLS (refer to Section II-C). TAO, however, adopts a weaker threat model compared to the standard logic locking threat model [8], [11], which is also adopted in this paper. The standard threat model assumes that an attacker can have access to: (1) a locked netlist, obtained through reverse engineering, and (2) a functional IC, purchased from the open market. TAO does not assume access to a functional IC and can protect against only a limited set of attacks that originate only at an untrusted foundry.

D. Contributions

In this paper, we demonstrate the feasibility of integrating HLS with state-of-the-art logic locking. Specifically, we integrate HLS with SFLL. SFLL-HLS analyzes the IR to determine the location of large (with 32 or 64 inputs) combinational units (e.g., adders and subtractors) in a design and locks these units using SFLL. We report the impact of locking on the overall system output in terms of SNR as well as the associated area, power, and timing overhead for an image processing application.

II. Background and Prior Work

A. SFLL

To achieve maximal resilience against SAT, removal, and approximate attacks, SFLL adopts the philosophy of stripped-functionality. It modifies the original circuit during design and later restores the functionality by applying the correct key. The locked circuit comprises a functionality-stripped circuit (FSC) and a restore unit, as illustrated in Fig. 1.

Functionality stripping can be implemented in various flavors such as SFLL-HD [15], SFLL-flex [15], or SFLL-fault [16]. In SFLL-HD, which represents the most effective flavor of SFLL, the output of the original circuit is modified for those input patterns that are of Hamming distance h from the secret key, i.e., $O\_locked \neq O, if(HD(IN, K_{secret}) \neq h)$ where K_{secret} is known only to the designer and HD denotes Hamming distance. Such input patterns are referred to as the protected input patterns. The restore operation is also based on the HD between the input pattern and the secret key. The FSC output is restored only when $HD(K, IN) = h$.

B. HLS: An overview

HLS interprets an algorithmic description of a design and creates RTL code that can be implemented in hardware [21], [22]. For example, HLS can translate a C/C++ code into Verilog code. The use of a higher abstraction level to represent a design speeds up the design as well as verification effort. HLS first compiles the high-level code into an IR, which is essentially a sequence of instructions (operations). The core HLS engine then bifurcates the operations into control and datapath operations. As illustrated in Fig. 2, the core HLS engine:

1) allocates the functional units to be used in the hardware implementation,
2) schedules the operations to be carried out in each clock cycle, and
3) binds each operation to a functional unit, while simultaneously adhering to the design constraints [23].

HLS is now supported by all major EDA vendors such as Intel and Xilinx. The vendors often provide HLS tools, e.g., Intel HLS Compiler and Xilinx Vivado HLS, along with libraries containing hardware components speahcific to certain families of FPGAs. Open-source HLS tools that target both FPGAs and ASICs (application-specific integrated circuits) are also available. LegUp is one of the widely adopted open-source HLS tools that has been evolving for more than a decade [22]. In addition to supporting traditional FPGA and ASIC flows, LegUp also supports a hardware/software co-design flow. In this flow, the operations are carried in part by dedicated hardware and in part by general-purpose processors.

C. Prior work on HLS + logic locking

Only a handful of research efforts investigate the link between HLS and logic locking. TAO is a recent technique that ”obfuscates” design functionality during high-level synthesis [20], albeit in a restrictive threat model. TAO assumes that the only untrusted entity is a foundry that cannot have access to a working chip. This threat model is applicable in only a handful of scenarios, e.g., in military ICs that are never available in the open market. Compared to TAO, SFLL-HLS adopts the standard logic locking threat model, which is applicable widely.

Built on the top of Bambu HLS framework [24], TAO can be considered as the integration of HLS and random logic locking [7]. The TAO algorithm operates on IR to identify and protect (using RLL) the following types of elements: arithmetic operations, constant values, and control flow. In most cases, the element to be protected is XORed with the secret key such that the correct output is obtained only upon application of the correct key.

III. SFLL-HLS

A. Architecture

SFLL-HLS aims to integrate HLS with SFLL (or other logic locking techniques) in a synergistic manner. As already mentioned, HLS tools allow faster design cycles by using a

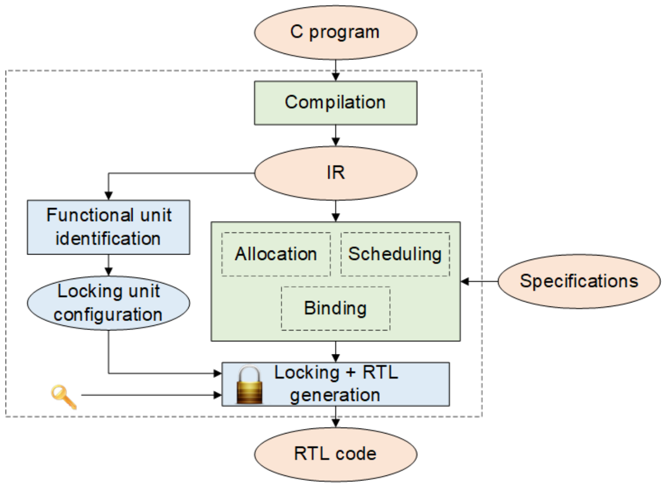
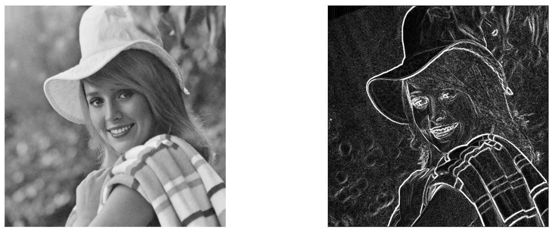


Fig. 2. The proposed SFLL-HLS flow, which extends the typical HLS flow. The entities shaded in blue are associted with logic locking.

higher level of abstraction. These tools, however, have not been designed to handle hardware security. SFLL-HLS attempts to bridge this gap and strives to incorporate security as another design objective into HLS.

Similar to the core HLS engine that operates on IR, SFLL-HLS also analyzes IR to identify the logic (functional units) to be protected. SFLL-HLS identifies only the combinational units, e.g., adders, multipliers, and subtractors, since SFLL is a combinational locking technique. A large key size (e.g., 64 or higher) is desired to achieve a reasonable security level. Accordingly, SFLL-HLS targets functional units with k or more inputs, where k denotes the key size. Upon determining the units-to-be-locked, their information (e.g., the output to be locked, the subset of inputs, the key size) is passed to the modified RTL code generation.

Recall from Fig. 1 that an SFLL-HD circuit comprises two restore units, one with a hard-coded key and the other with key inputs that are driven by a tamper-proof memory. The modification to the RTL generation unit involves creating the code for two restore units and replacing the original functional units with their locked counterparts. Fig. 2 presents the flow of SFLL-HLS; the entities associated with logic locking are shaded blue.



(a) (b)

Fig. 3. (a) Original image, (b) Output of the sobel edge detector obtained through simulation using Modelsim.

B. Case study

We demonstrate the application of SFLL-HLS on an image processing example obtained from [25]. The example is about edge detection, and more specifically, about applying the Sobel filter to an image. Sobel filtering is the convolution of an image with a pair of 3x3 kernels/filters [25]. The kernels are typically referred to as Gx and Gy; they correspond to detecting horizontal and vertical edges, respectively. Fig. 3 presents an example image and the output of the Sobel edge detector.

Experimental setup. All experiments are conducted using the open-source LegUp HLS tool [22], Modelsim, and Matlab. The Sobel edge detector code is written in C. The input image is specified as a char array, which translates into a ROM in the hardware implementation. In this demonstrative case study, we do not employ any pipe-lining or loop unrolling. The original Sobel edge detector circuit is synthesized using the typical HLS flow, highlighted in green color in Fig. 2. The circuit is then locked following the proposed SFLL-HLS flow. An analysis of the IR generated by the compiler reveals that there are multiple add/sub instructions operating on 32-bit data. Accordingly, we configure the RTL generation block to generate RTL code to lock one of these adders with a 32-bit key. To compute area, power, and timing overhead, we synthesize the original and the locked circuits using Synopsys DC Compiler in conjunction with the NanGate Open Cell Library. The image ROM and the RAM used for storing intermediate results are not considered in the overhead computation..

Signal to noise ratio (SNR). Fig. 4 illustrates the effect of locking on the edge detector output. The figure displays the error (difference) between the correct output the edge detector (shown in Fig. 3(b)) and the "incorrect" output, obtained for a random incorrect key for different values of HD h. The darker the image, the lower the error. One would expect the error to increase with h. However, since the locked adder drives only an internal register, the effect of the incorrect output may not always be propagated to the output. We observe that for h=8, the error is almost negligible as indicated by an SNR value of 83 and a completely dark image in Fig. 4(b). The same trend can also be observed in Table II, which reports the SNR (computed using the Matlab PSNR function) between the locked circuit output and correct output. The error introduced

TABLE II
SNR OF THE LOCKED SOBEL EDGE DETECTOR FOR DIFFERENT VALUES OF HAMMING DISTANCE.

h	**4**	**8**	**12**	**16**
SNR	42	83	43	52

TABLE III
AREA, POWER, AND IMPLEMENTATION COST OF SOBEL EDGE DETECTOR. THE REPORTED VALUES ARE AVERAGED OVER FOUR DIFFERENT VALUES OF H.

Parameter	**Baseline**	**Locked**	**Overhead (%)**
Area (μm^2)	12933	13212	2.2
Power (μW)	1269.5	1407.3	10.9
Timing (ns)	3.37	3.37	0

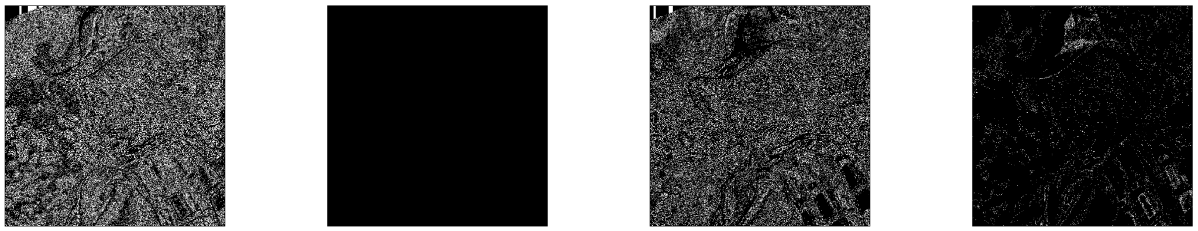

(a) h=4 (b) h=8 (c) h=12 (d) h=16

Fig. 4. Error between the correct output and the incorrect output of the locked Sobel edge detector for different values of h. The corresponding SNR values are 42, 83, 43 and 52, respectively.

is almost identical for h=4 and h=8. This also emphasizes the intricacies of locking internal components; the impact on the overall system output may be different from the expected one.

Implementation overhead. Table III reports the area, power, and timing overhead of the locked circuit, which is 2.2%, 10.9% and 0%, respectively. Note that the power overhead can be traded-off with the area overhead by enabling loop unrolling or pipe-lining.

Acknowledgement

The work was supported by in part by the Office of Naval Research (ONR Award #N00014-18-1-2058), the National Science Foundation (NSF CNS-1749175), and the the Defense Advanced Research Projects Agency (DARPA HR001116S0001-FP39).

IV. Conclusion

The paper presents SFLL-HLS that integrates logic locking and high-level synthesis with the goal of enabling system-level security. An image processing case study successfully demonstrated the locking of a Sobel edge detector in conjunction with the LegUp HLS tool. The paper shed light on various intricacies involved in the integration of HLS with SFLL and emphasizes the need for further research in this area.

References

[1] Evertiq, "Top 10 fabless IC design companies Qualcomm in the lead," https://evertiq.com/news/40662, 2016, last accessed on 02/09/19.

[2] R. Torrance and D. James, "The State-of-the-Art in Semiconductor Reverse Engineering," *IEEE/ACM Design Automation Conference*, pp. 333–338, 2011.

[3] Y. Alkabani and F. Koushanfar, "Active Hardware Metering for Intellectual Property Protection and Security," *USENIX Security Symposium*, pp. 291–306, 2007.

[4] A. Kahng, J. Lach, W. H. Mangione-Smith, S. Mantik, I. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe, "Watermarking Techniques for Intellectual Property Protection," *IEEE/ACM Design Automation Conference*, pp. 776–781, 1998.

[5] J. Rajendran, M. Sam, O. Sinanoglu, and R. Karri, "Security Analysis of Integrated Circuit Camouflaging," *ACM SIGSAC Conference on Computer & Communications Security*, pp. 709–720, 2013.

[6] R. Jarvis and M. McIntyre, "Split Manufacturing Method for Advanced Semiconductor Circuits," 2007, US Patent no. 7,195,931.

[7] J. Roy, F. Koushanfar, and I. Markov, "Ending Piracy of Integrated Circuits," *IEEE Computer*, vol. 43, pp. 30–38, 2010.

[8] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Security Analysis of Logic Obfuscation," *IEEE/ACM Design Automation Conference*, pp. 83–89, 2012.

[9] M. Yasin, B. Mazumdar, J. Rajendran, and O. Sinanoglu, "SARLock: SAT Attack Resistant Logic Locking," *IEEE International Symposium on Hardware Oriented Security and Trust*, pp. 236–241, 2016.

[10] T. S. Perry, "Why Hardware Engineers Have to Think Like Cybercriminals, and Why Engineers Are Easy to Fool," http://spectrum.ieee.org/view-from-the-valley/computing/embedded-systems/why-hardware-engineers-have-to-think-like-cybercriminals-and-why-engineers-are-easy-to-fool, 2017.

[11] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the Security of Logic Encryption Algorithms," *IEEE International Symposium on Hardware Oriented Security and Trust*, pp. 137–143, 2015.

[12] Y. Xie and A. Srivastava, "Mitigating SAT Attack on Logic Locking," *International Conference on Cryptographic Hardware and Embedded Systems*, pp. 127–146, 2016.

[13] M. Yasin and B. Mazumdar and O. Sinanoglu and J. Rajendran, "Removal Attacks on Logic Locking and Camouflaging Techniques," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2018.

[14] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, "AppSAT: Approximately Deobfuscating Integrated Circuits," *IEEE International Symposium on Hardware Oriented Security and Trust*, pp. 95–100, 2017.

[15] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. Rajendran, and O. Sinanoglu, "Provably-Secure Logic Locking: From Theory To Practice," *ACM SIGSAC Conference on Computer & Communications Security*, pp. 1601–1618, 2017.

[16] A. Sengupta, M. Nabeel, M. Yasin, and O. Sinanoglu, "ATPG-based cost-effective, secure logic locking," *IEEE VLSI Test Symposium*, pp. 1–6, 2018.

[17] R. Chakraborty and S. Bhunia, "HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1493–1502, 2009.

[18] M. Fyrbiak, S. Wallat, J. Déchelotte, N. Albartus, S. Böcker, R. Tessier, and C. Paar, "On the difficulty of fsm-based hardware obfuscation," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 293–330, 2018.

[19] M. Yasin and O. Sinanoglu, "Evolution of Logic Locking," *IFIP/IEEE International Conference onVery Large Scale Integration*, pp. 1–6, 2017.

[20] C. Pilato, F. Regazzoni, R. Karri, and S. Garg, "TAO: Techniques for Algorithm-Level Obfuscation during High-Level Synthesis," *Proceedings of the 55th Annual Design Automation Conference*, pp. 155:1–155:6, 2018.

[21] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin, *HighLevel Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[22] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems," *ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 33–36, 2011.

[23] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[24] C. Pilato and F. Ferrandi, "Bambu: A Modular Framework for the High Level Synthesis of Memory-Intensive Applications," *International Conference on Field programmable Logic and Applications*, pp. 1–4, 2013.

[25] LegUp, "LegUp High-Level Synthesis Tutorial: Sobel Filtering for Image Edge Detection," https://www.legupcomputing.com/static/downloads/tutorials/Sobel_Tutorial.pdf, 2016, last accessed on 07/09/19.