Investigating the Effect of different eFPGAs fabrics on Logic Locking through HW Redaction

Chaitali Sathe, Yiorgos Makris and Benjamin Carrion Schafer
Department of Electrical and Computer Engineering
The University of Texas at Dallas, TX, USA
{ChaitaliGajanan.Sathe,yiorgos.makris,schaferb}@utdallas.edu

2022 IEEE 15th Dallas Circuit And System Conference (DCAS) | 978-1-6654-9885-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/DCAS53974.2022.9845535

***Abstract*—Most VLSI design companies are now fabless. This implies that they need to rely on third party fabs to fabricate their Integrated Circuits (ICs). Because these fabs are often located in geographical different locations it is important to have mechanisms in place to protect these companies against Intellectual Property (IP) theft. One approach that has become very popular due to its relative simplicity and practicality is logic locking. One of the problems with traditional locking mechanisms is that the locking circuitry is built into the netlist that the VLSI design company delivers to the foundry which has now access to the entire design including the locking mechanism. This implies that they could potentially tamper with this circuitry or reverse engineer it to obtain the locking key. One relatively new approach that has been coined logic locking through *omission*, or hardware redaction, maps a portion of the design to an embedded FPGA (eFPGA). The bitstream of the eFPGA now acts as the locking key. This new approach has been shown to be more secure as the foundry has no access to the bitstream during the manufacturing stage. The obvious drawbacks are the increase in design complexity and the area and performance overheads associated with the eFPGA. In this work investigate the trade-offs of mapping different portions of behavioral descriptions for High-Level Synthesis using two different eFPGA fabrics and show that it is important to choose the eFPGA fabric based on the characteristic of the design to be locked.**

***Index Terms*—Functional Locking, Behavioral IP, High-Level Synthesis. embedded FPGAs**

I. Introduction

THE semiconductor industry has been undergoing a significant transformation in the last two decades. In the past, companies were vertically integrated and designed, verified a nd m anufactured c ompletely i n-house t heir Integrated Circuits (ICs). In the early stages they even designed their own Electronic Design Automation (EDA) tools. This has rapidly changed due to the increase in complexity of designing today's multi-billion transistors ICs. There are only a few companies left that can continue to do the entire VLSI design process in-house (i.e., Intel or Samsung Electronics). Most semiconductor companies are now fabless and rely extensively on third party IPs (3PIPs). Moreover, they often outsource significant portions of the design efforts to external third party companies, e.g., the physical design stage and/or verification. This leaves these companies extremely vulnerable to multiple security threats. Some of these include the malicious alteration of their hardware to include Hardware Trojans. Other threats include the ability of third parties to *steal* the IP developed by these fabless companies and which represent their main value added. It is therefore extremely important to introduce efficient methods to enable these companies to protect their IPs.

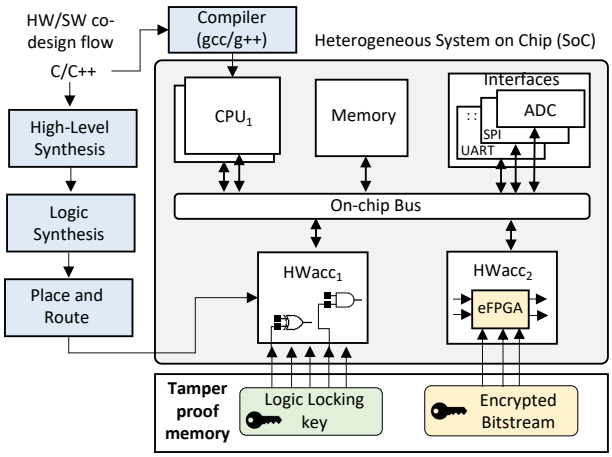

Fig. 1. Overview of typical logic locking mechanisms (through additional locking gates and through eFPGAs) in the context of modern heterogeneous SoCs.

Fig. 1 shows an example of a typical heterogeneous SoC that is composed of multiple embedded processors, embedded memory, interfaces and a variety of hardware accelerators. The number and implementation of these accelerators are often the main differentiating factor between different SoC offerings from different companies as the rest of the components are standard off-the-shelf IPs that can be sourced from third party vendors (e.g., ARM processor). The figure also shows a typical VLSI design flow starting from the application level that needs to be partitioned in hardware (HW) and software (SW).

It is therefore particularly important to protect these hardware accelerators from being reversed engineered. One promising approach that is being widely investigated is logic locking and design obfuscation [1]. In traditional logic locking, additional *locking* gates are added to the circuit. A logic locking key is in turn stored in a tamper-proof memory. Without the correct key, the locked circuit does not work as specified (either the output is incorrect, or the performance is degraded). In the example in Fig. 1, the logic locking circuitry in HW accelerator 1 ($HWacc_1$) is composed of an XOR gate and an AND gate. For this circuit to operate as

978-1-6654-9885-2/22/$31.00 ©2022 IEEE

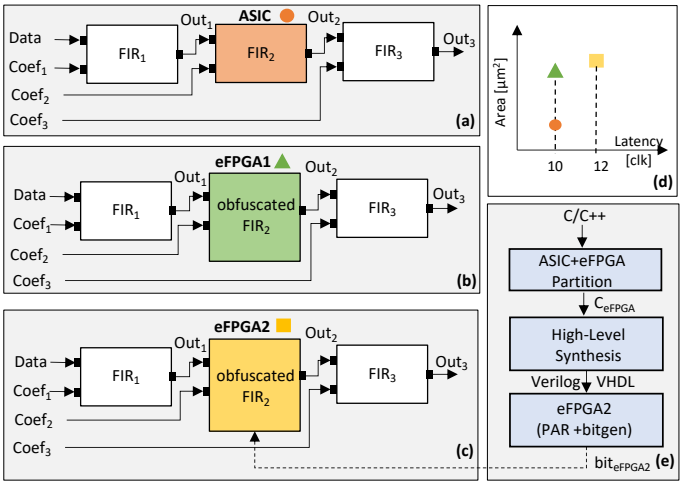

Fig. 2. Example of a three-stage decimation filter. (a) Original ASIC-only implementation; (b) Obfuscated design with FIR_2 mapped to eFPGA1; (c) Same obfuscated design with FIR_2 mapped to eFPGA2; (d) Area vs. latency trade-offs of different implementations; (e) Overview of partition flow

expected, the key input to the XOR gate should be logic 0 and the key to the AND gate input set to logic 1. Fig. 1 also shows a relatively new way to lock hardware circuits through *omission*. In this case, a portion of $HWacc_2$ is mapped to an eFPGA. The eFPGA bitstream now acts as the logic locking key and the un-programmed eFPGA design is sent to the foundry for fabrication. This approach has been shown to be more secure as the search space is much larger than traditional locking methods considering that the bitstream not only determines the logic function mapped onto the eFPGA but also the interconnect [2], [3] making this approach even resilient against modern SAT attacks [4]. Thus, mapping a smaller portion of a design to an eFPGA can serve as a strong locking mechanism.

One of the main problems with this approach is that the overheads associated with using eFPGAs is significant. The authors in [5] reported a 10× area, delay and power overheads of FPGAs vs. ASIC designs. It is therefore important to develop a framework that can minimize these overheads. Moreover, considering that there are FPGAs with different granularities available, the question that we aim to address in this work is if some eFPGA are better suited than others for logic locking . In summary, the main contributions of this work are:

- Introduce an automated flow that partitions untimed behavioral descriptions for HLS into an ASIC part and an eFPGA to lock the final circuit.
- Present extensive experimental comparing different eFPGA fabrics overheads.

II. Motivational Example

Fig. 2 shows a motivational example for this work. In this particular case a 3-stage decimation filter which is composed of cascading three Finite Impulse Response (FIR) filters, where the output of FIR_1 is passed as an input to the second FIR filter (FIR_2) and the output of this second filter passed to the third FIR filter (FIR_3). Fig. 2(a) shows the original unprotected ASIC implementation. Fig. 2(b) shows one possible obfuscation partitioning, which consists of mapping one of the FIR filters (in this case FIR_2) to an eFPGA. This design is now protected against reverse engineering when sent to an untrusted fab as the fab does not have access to the bitstream that configures the eFPGA portion of the design (the bitstream is not sent to the fab).The entire design is hence, partitioned into an ASIC portion that contains FIR_1 and FIR_3 and an eFPGA portion which contains FIR_2: Design=ASIC(FIR_1, FIR_3) ∪ eFPGA(FIR_2). Fig. 2(d) shows the area overhead introduced by this obfuscation approach when using different eFPGA fabric as compared to the ASIC only approach

Fig. 2(e) show our partitioning approach. The idea is partition an untimed behavioral description for HLS into an ASIC part and an eFPGA part. Based on the eFPGA fabric used different area and performance overheads will be achieved

III. Related Work

Logic obfuscation can be describe as the process that transforms a circuit into another functional equivalent version that is significantly, ideally impossible, to reverse engineer. This research area has recently received significant attention due to the importance of this topic, especially considering that most semiconductor companies are now fabless.

Some of this research includes logic encryption [6], active metering [7], state obfuscation [8], split manufacturing [9] and design camouflaging [10]. Each of these proposed solutions has its strengths and weaknesses with no single solution successfully addressing the security and trust challenges in a cost-effective manner.

One common problem though with most of these approaches is that the fab still has access to the entire circuit including the obfuscated logic (except for split manufacturing). Thus, to address this, one relatively new approach has been to selectively extract a small portion of the circuit and mapping it to an eFPGA. This allows designers to *hide* a portion of their design and make the circuit unusable without the correct eFPGA bitstream. To the best of our knowledge, this approach was first introduced in [2], where the authors present a dedicated eFPGA fabric that they call TRAP to minimize the overhead introduced by conventional eFPGAs. Although the authors showed that this approach works well, the dedicated fabric is only usable to hide simple combinational logic. In this work the authors also present a partitioning flow for RTL descriptions. Bo et al. [3] proposed a similar partitioning methodology but raising the level of abstraction from the RT-level to the behavioral level and presented an automated partitioning flow for behavioral descriptions for HLS. In [11] the authors studied how to reduce the area overhead introduced by the eFPGA by using runtime reconfigurable coarse grain FPGAs. The same approach was used in [4] to obfuscate portions of the RISC-V control path. Finally, this ASIC+eFPGA flow was also shown to be effective to *hide* implementation details of two functionally equivalent designs, e.g., ANN activation functions [12].

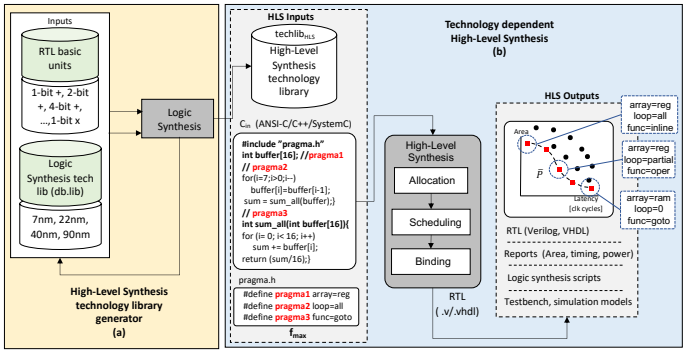

Fig. 3. Typical HLS Flow (a) Technology library generation flow; (b) HLS inputs, steps and outputs.

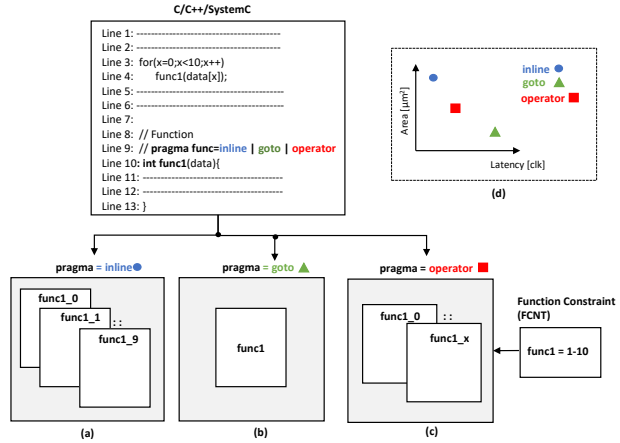

Fig. 4. HLS options to synthesize functions. (a) Inline, (b) goto or (c) functional operator. (d) Trade-offs between the three different ways to synthesize functions

To the best of our knowledge this is the first study that analysis multiple eFPGA fabrics and presents an automated flow to evaluate their overheads.

IV. Threat Model

This work considers the same threat model used in most previous functional locking work and assumes that any party involved in the design and fabrication of the circuit is a threat and in particular the fab that can reverse engineer any traditional circuit given as GDSII netlist. We assume that the attacker has access to the layout and significant resources and technical knowledge in reverse engineering. The main goal of the attacker is to reverse engineer the IC to sell it as a pirate copy or to acquire the IP of the IC for its own profit. The attacker also has access to an oracle, which is a fully functional IC obtained legally from the market. The attacker can also apply input patterns to the locked IC and observe its response.

V. High-Level Synthesis

Before we proceed describing the proposed work, it is important to review what HLS is and how it works. Fig. 3 shows an overview of the complete HLS process. HLS can be described as a process to convert untimed behavioral descriptions into efficient hardware that implements that behavior. The input to the HLS process, as shown in Fig. 3 (b), are the behavioral description to be synthesized in e.g., ANSI-C, C++ or SystemC, a technology library ($techlib_{HLS}$) that contains the area and delay information of basic operations (i.e., adders and multipliers of different bitwidth), a target synthesis frequency (f_{max}) and a set of synthesis directives in the form of pragmas (pragma.h). These synthesis directives are extremely important as they allow the designer to control the synthesis process. In particular, these directives control how to synthesize arrays (RAM or registers), loops (unroll, partially unroll, not unroll or pipeline) and functions (inline or not). In the example shown in Fig. 3(b) the code snippet contains one array and two loops.

The HLS process then parses the behavioral description and constraints and performs three main steps: (1) resource allocation, (2) scheduling and (3) binding. In the resource allocation stage the number and type of hardware resources from $techlib_{HLS}$ are extracted. In the scheduling phase the different operations in the behavioral description are assigned to individual clock steps based on the number of available resources and finally, in the binding stage the hardware resources are bound to different operations in the scheduled operations.

To achieve high quality RTL, the HLS process needs to know that accurate delay of the different functional units (FUs) that are mapped to different operations in the behavioral description. For this, commercial HLS tools provide a library characterizer shown in Fig. 3 (a) that generates the technology library for the HLS process. This library characterizer synthesizes (logic synthesis) different basic units with different bitwidths. Basically the library characterizer generates automatically the RTL code of these basic units, synthesizes it with the target technology specified by the user, which has to match the technology used during HLS and back-annotates the area and delay information reported by the logic synthesis tools into the HLS technology library ($techlib_{HLS}$). This process needs to be executed before the HLS process is executed and although time consuming, it only needs to be execute once. It should be noted that FPGA vendors do not provide this flow as they pre-generate these technology libraries for their particular FPGAs and include them with their HLS tool.

The output of the HLS process is the RTL code (Verilog or VHDL) and a set of reports that summarize the area and performance of the synthesized circuit. Commercial HLS tools also generate synthesis scripts to interface the HLS tool with the logic synthesis process and testbenches to verify the generated circuit as shown on Fig. 3(b).

As mentioned previously, commercial HLS tools make extensive use of synthesis directives to mainly decide how to synthesize arrays, loops and functions. In this work we will use a synthesis directive to encapsulate the function to be mapped onto the eFPGA as an operator. Fig. 4 shows an example of the three main ways to synthesize a function in HLS. The first is to inline functions where the body of the function is copied at the function call point (Fig. 4(a)).

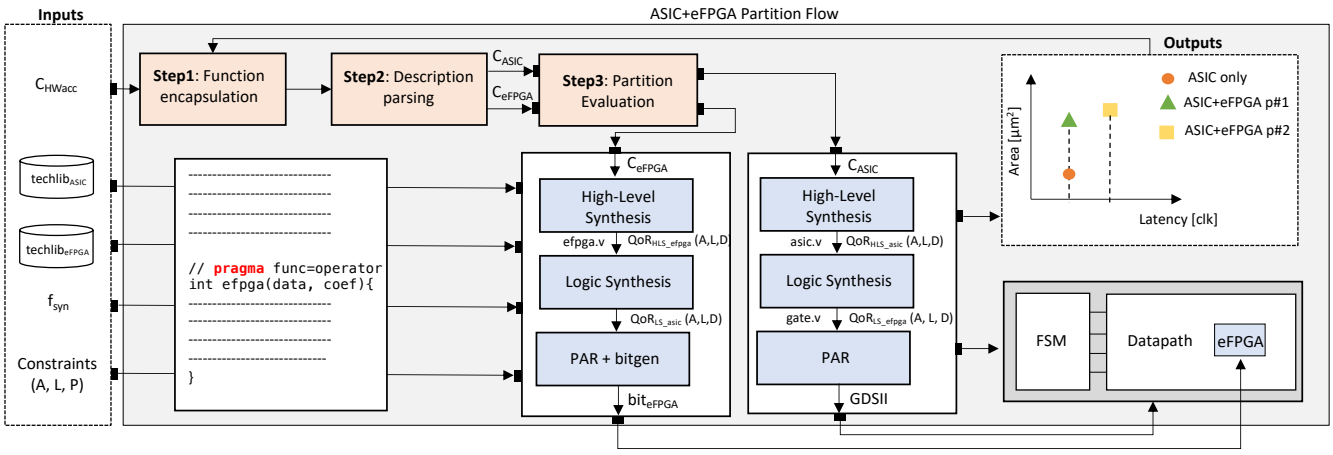

Fig. 5. Overview of complete ASIC+eFPGA partitioning flow.

Another way to synthesize functions is as a single hardware block (goto). This will make the resultant HW circuit slower, but smaller compared to inlining the function (Fig. 4 (b)). One alternative way is to encapsulate the function as an operator as shown in Fig. 4(c). In this case the HLS tool generates a constraint file that allows the designer to specify now many functions to be instantiated. As shown in Fig. 4(d), using these different synthesis directives lead to different area vs. performance trade-offs. In this work we use the operator option to as this option encapsulates the function as a separate module and allows to synthesize the function with its own constraints. This is used to delimit the partition between the portion of the behavioral description to be mapped to the ASIC and the portion to be mapped to the eFPGA, which is basically the encapsulated function.

VI. Embedded FPGAs

The obfuscated portions of a system designcan be mapped to any embedded FPGA (eFPGA). There are multiple commercial eFPGAs on the market. The most notable vendors include Achronix [13], Menta [14] and Quicklogic [15]. These eFPGAs are based on traditional multi-input, 1-output LUTs and also contain multiple hard macros such as embedded DSP modules and memories.

Another eFPGA fabric that was recently introduced is a transistor-level fabric called Transistor-Level Programmable Fabric (TRAP) [16]. The TRAP fabric consists of hierarchically arranged CMOS transistors. Basically these transistors can be stitched together into gates or state-holding components. For better performance and area efficiency of the fabric, an element also contains a built-in flip-flop (DFF), a full adder (FA) and a multiplexer (MUX). This could be considered a ultra-fine grain FPGAs

On the other side of the spectrum we could also use coarse-grained FPGAs like Renesas Technology Stream Transpose Processor (STP) [17]. The STP is composed of tiles onto which the datapath of an application is mapped. Each tile in turn consists of an array of 8×8 Processing Elements (PEs). Each tile is also surrounded by embedded memory and dedicated DSP blocks. A state transition controller (STC) that is located in the center of the STP is responsible for reconfiguring the tiles with the correct functionality every clock cycle. This reconfiguration takes 1ns. In [18] the authors showed that this fabric can be efficiently used for logic locking due to its ability to reconfigure itself, hence, reducing the area overhead.

In this work we will focus on the first two eFPGA fabrics as they are the most commons one and leave the comparison with CGRAs for future work.

VII. Proposed Method Overview

Fig. 5 depicts graphically the overall flow of the proposed method. The framework takes as input the untimed behavioral description to be locked (C_{HWacc}), the technology libraries for the target ASIC ($techlib_{ASIC}$) and eFPGA ($techlib_{eFPGA}$), and the target synthesis frequency (f_{syn}). The user can also specify a set of constraints as inputs in terms of the total area (A), latency (L) and power (P) that the final circuits shown not exceed. The method is composed of three main steps. described in detailed below.

Step 1 - Function Encapsulation: This first step encapsulates portions of the behavioral description into functions. The previously described pragma to encapsulate this function as an operator is then specified to tell the HLS tool that the contents of the function should be synthesized as a black box. This allows a clean and easy way to partition the design into the ASIC and eFPGA that is fully transparent to commercial HLS tools.

Step 2 - Description Parsing: This second step takes as input partitioned behavioral description and parses it. The HLS tools automatically generates two separate files (C_{ASIC} and C_{eFPGA}) that can now be synthesizes separately with different constraints and more importantly technology libraries (ASIC and eFPGA).

Step 3 - Partition Evaluation: This last step synthesizes the two behavioral description (C_{ASIC} and C_{eFPGA}) using the same target synthesis frequency (f_{syn}), as the final circuit still needs to meet the original, ASIC only maximum delay

constraint. Each behavioral description as shown in Fig. 5 is synthesized using the particular technology library. After HLS we can already understand the overheads associated with the generated partition, as the HLS tools generates a Quality of Report (QoR) file with the total area in case of ASIC design and resources utilization for FPGA, estimated delay and latency. In case of FPGA area, since we don't have access to the tools to calculate exact area, the LUTs and DSPs were converted into μm^2 following the indications of [19] which approximately calculates FPGA tile areas including routing area. For more accurate results the generated RTL can be further synthesized and/or placed and routed. For TRAP, we calculated area and size of the bitstream accurately as indicated in [2].

This step allows to quantify the overhead of this particular partition quickly. This process can then be repeated with a new partition until a given exit condition is reached, e.g., the given overall constraints are met. Although the number of partitions might seem significant and hence, the running time to find the best partition, in reality there is only a very limited number of partitions as e.g., if-conditions and switch-cases need to be grouped together, restricting significantly the number of valid partitions. In order to find highly secured partitions, 1) it must utilize large numbers of FPGA resources- LUTs or DSPs, as use of large numbers of LUTs, DSPs weakens any SAT-based attack or brute force attack by increasing the search space. 2) Obfuscated function logic must be complex, irregular, diverse and not easy to guess from behavior of rest of the circuit. While high numbers of FPGA resources leads to high security but it causes area, also obfuscated part of the circuit must play very crucial role in performance and overall response of the circuit to make this technique highly effective.

VIII. Security Analysis

The proposed locking mechanism leads to a more robust functional locking in two main ways: through the increase in the search space, thus, making SAT attacks virtually impossible and secondly through resource sharing of the locking logic, thus, making removal attacks not possible.

Protection against SAT Attack: First it increases the search space, by forcing the attacker to run a SAT attack on each of the unique FSM states. Moreover, because the order of the correct keys is needed, it has to run this attack on all key permutations. This implies that for a circuit with n FSM states, $n!$ distinct combination of key sequences can be generated, as opposed of just one single key used in previous work.

Protection against Removal Attack: In addition, most of the previous locking mechanisms are subject to removal attacks. These attacks identify the circuitry responsible for the logic locking and removes it at the foundry. In our case, because this logic is fully shared with the rest of the circuitry, it is impossible to remove. The nature of the HLS process, leads to maximize resource sharing. In resource sharing a single functional unit is shared among different operations in the source code by inserted multiplexers at its inputs and outputs. The Finite State Machine (FSM) created by the HLS synchronizes all the multiplexers' control signals in order to steer the data through the datapath containing the shared FUs. Thus, the RTL code generated through HLS typically shared most of its resources when possible. This makes our approach extremely robust against removal attacks. We measure security in terms of the time-to-break (TTB) as indicated in [3] where TTB is a function of bitstream. Bitstream size for TRAP based eFPGA is calculated from number of instances mapped on TRAP [2]. Since bitstream size for any commercial FPGA is constant for given FPGA model, we deduce size of the bitstream from number of resources utilized. From high level architecture of ALM in Cyclone V we know that ALM is made of two 6-input LUT, eight 2:1 MUXs, two internal adders and four registers. For experimental purpose we conservatively assume that one ALM produces bitstream of size 136 bits. Figure 6 shows relation between bitstream size and TTB in years for brute force attack using single FPGA model and it also shows the threshold for number of resources to get higher security.

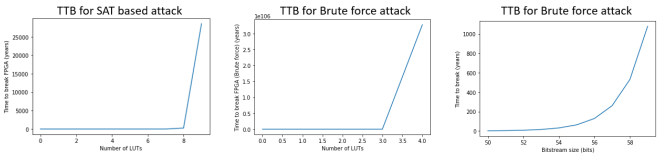

Fig. 6. (a) TTB for SAT based attack on Cyclone V eFPGA for number of LUTs, (b) TTB for brute force attack on Cyclone V eFPGA for number of LUTs, (c) TTB for brute force attack on TRAP eFPGA for given bitstream size

IX. Experimental Results

Five computationally intensive applications from the open source S2Cbench SystemC benchmarks suite [20], were used to test our proposed flow. These benchmarks comply with the latest Accelera's Synthesizable SystemC subset and hence, can be synthesized with any commercial HLS tools as these all support SystemC. *FIR* filter, *aes* a block cipher, *snow3G* stream cipher, *kasumi* block cipher used in mobile communication, *adpcm* as delta pulse code modulator. Table I summarizes the main characteristics of the benchmarks used in terms of their number of functions can be mapped on the eFPGA and possible designs. In this work we use NEC's CyberWorkBench [21]. The target synthesis frequency is set in all cases to 100MHz. Global Foundries 12LP library is targeted for TRAP and all areas are scaled to 12nm technology. This will be our base-line, ASIC-only (*ASIC*), design against which we will measure the overheads in terms of area of the proposed locking mechanism . Delay is not used as the HLS process always guarantees to meet the given target synthesis frequency by inserting additional clock steps. Two eFPGA fabrics are compared. We use a Cyclone V FPGA (*LUT*)to replicate a regular island-based fine grain eFPGA and also the transistor level eFPGA (*TRAP*).

We have performed an exhaustive enumeration of all the partitions and report the partitions that have the smallest

TABLE I
BENCHMARK CHARACTERISTICS

	Benchmarks				
	FIR	aes	snow3G	kasumi	adpcm
#Functions	1	6	5	3	2
#Total designs	2	27	8	88	4

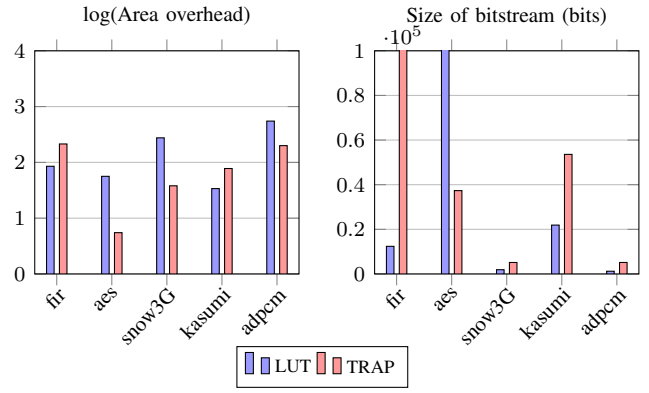

Fig. 7. Area overhead and size of bitstream for LUT style FPGA (LUT) and transistor-level eFPGA (TRAP).

overheads that would make the design secure in Fig.7. In this context we assume that the partitioned design is secure if the attacker would need over one year of time to break the system by generating different valid bitstreams as shown in Fig. 6.

From the results we can make the following observations:
Observation 1: The eFPGA fabric used has a significant impact on the area overheads. In particular from Fig. 7 we can observe that for benchmark *FIR* the LUT-style fabric leads to an average area overhead of $10^{1.93}$, while the TRAP eFPGA leads to an average area overhead of $10^{2.33}$. This difference in area overheads can be explained with the architectural difference between TRAP and LUT based FPGA. Benchmark such as *FIR* has multiply and accumulate operations which use embedded DSPs modules while TRAP eFPGA doesn't have any embedded modules. Other benchmarks which don't use DSPs have area overheads comparable to each other.
Observation 2: For instances which dominantly use basic logic operations than complex mathematical operations need more numbers of ALUTs to map the operations than TRAP instances as TRAP instances are made of standard CMOS logic cells. In *AES* cipher banchmark, the function mapped on eFPGA mostly has basic EX-OR and AND operations because of that TRAP shows very small area overhead than LUT based eFPGA.
Observation 3: FPGA area approximation used here [19] does not evaluates area occupied by I/O ports but TRAP CAD tools evaluate exact area occupied by entire circuit including I/O and routing [2]. The benchmark *Kasumi* has higher number of I/O ports than others and hence explains the difference in area overhead.

In summary we can conclude that the proposed approach works well when partitioning an untimed behavioral description for HLS into an ASIC and an eFPGA part and that it allows us to quickly measure the area overheads and security associated with any partition.

X. CONCLUSIONS

In this work we have proposed an automatic method to functionally lock behavioral IPs for HLS by mapping a portion of the untimed behavioral description to an eFPGA. We proposed a fully automatic flow that is transparent to commercial HLS tools by encapsulating the portion to be mapped to the eFPGA as a function annotating a synthesis directive to synthesize it as an operator. This allows us to compare the different overheads associated with this locking mechanisms when using different eFPGA fabrics.

XI. ACKNOWLEDGEMENTS

This work is partially supported by the NSF Industry/University Cooperative Research Center on Hardware and Embedded Systems Security and Trust (CHEST) through project #P7_20.

REFERENCES

[1] M. T. Rahman *et al.*, "Defense-in-depth: A recipe for logic locking to prevail," *Integration, the VLSI Journal*, vol. 72, pp. 37–57, Jan 2020.
[2] M. M. Shihab *et al.*, "Design obfuscation through selective post-fabrication transistor-level programming," in *DATE*, 2019, pp. 528–533.
[3] B. Hu *et al.*, "Functional obfuscation of hardware accelerators through selective partial design extraction onto an embedded fpga," in *GLSVLSI*, 2019, p. 171–176.
[4] P. Mohan *et al.*, "Hardware redaction via designer-directed fine-grained efpga insertion," in *DATE*, 2021, pp. 1186–1191.
[5] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE TCAD*, vol. 26, no. 2, pp. 203–215, 2007.
[6] J. Rajendran *et al.*, "Fault analysis-based logic encryption," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 410–424, Feb 2015.
[7] J. A. Roy *et al.*, "Epic: Ending piracy of integrated circuits," in *DATE*, 2008, pp. 1069–1074.
[8] R. S. Chakraborty and S. Bhunia, "HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection," *IEEE TCAD*, vol. 28, no. 10, pp. 1493–1502, Oct 2009.
[9] J. Rajendran *et al.*, "Is split manufacturing secure?" in *2013 DATE*, 2013, pp. 1259–1264.
[10] ——, "Security analysis of integrated circuit camouflaging," in *SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, 2013, pp. 709–720.
[11] J. Chen and B. Carrion Schafer, "Area efficient functional locking through coarse grained runtime reconfigurable architectures," in *ASP-DAC*, 2021, p. 542–547.
[12] J. Chen *et al.*, "DECOY: DEflection-Driven HLS-Based Computation Partitioning for Obfuscating Intellectual PropertY," in *DAC*, 2020, pp. 1–6.
[13] Achronix, "Speedcore embedded fpga ip," 2022.
[14] Menta, "https://www.menta-efpga.com," 2022.
[15] Quicklogic, "ArticPro, https://www.quicklogic.com," 2022.
[16] J. Tian *et al.*, "A field programmable transistor array featuring single-cycle partial/full dynamic reconfiguration," in *DATE*, 2017, pp. 1336–1341.
[17] Renesas, "STP, http://www.renesas.com/products/soc/asic/programmable," 2021.
[18] J. Chen and B. C. Schafer, "Area efficient functional locking through coarse grained runtime reconfigurable architectures," ser. ASPDAC '21, 2021, p. 542–547.
[19] H. Wong, V. Betz, and J. Rose, "Comparing FPGA vs. Custom CMOS and the Impact on Processor Microarchitecture," in *FPGA*, 2011, p. 5–14.
[20] B. Carrion Schafer and A. Mahapatra, "S2CBench:Synthesizable SystemC Benchmark Suite," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 53–56, 2014.
[21] NEC CyberWorkBench. (2019). [Online]. Available: www.cyberworkbench.com