Functional Locking through Omission: From HLS to Obfuscated Design

Zi Wang, Shayan Omais Mohammed, Yiorgos Makris and Benjamin Carrion Schafer
Department of Electrical and Computer Engineering
The University of Texas at Dallas, TX, USA
{zi.wang5,shayan.mohammed,yiorgos.makris,schaferb}@utdallas.edu

2021 IEEE 39th International Conference on Computer Design (ICCD) | 978-1-6654-3219-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCD53106.2021.00095

Abstract*—VLSI design companies are now mainly fabless and spend large amount of resources to develop their Intellectual Property (IP). It is therefore paramount to protect their IPs from being stolen and illegally reversed engineered. The main approach so far to protect the IP has been to add additional locking logic such that the circuit does not meet the given specifications if the user does not apply the correct key. The main problem with this approach is that the fabless company has to submit the entire design, including the locking circuitry, to the fab. Moreover, these companies often subcontract the VLSI design back-end to a third-party. This implies that the third-party company or fab could potentially tamper with the locking mechanism. One alternative approach is to lock through omission. The main idea is to judiciously select a portion of the design and map it onto an embedded FPGA (eFPGA). In this case, the bitstream acts as the logic key. Third party company nor the fab will, in this case, have access to the locking mechanism as the eFPGA is left un-programmed. This is obviously a more secure way to lock the circuit. The main problem with this approach is the area, power, and delay overhead associated with it. To address this, in this work, we present a framework that takes as input an untimed behavioral description for High-Level Synthesis (HLS) and automatically extracts a portion of the circuit to the eFPGA such that the area overhead is minimized while the original timing constraint is not violated. The main advantage of starting at the behavioral level is that partitioning the design at this stage allows the HLS process to fully re-optimize the circuit, thus, reducing the overhead introduced by this obfuscation mechanism. We also developed a framework to test our proposed approach and plan to release it to the community to encourage the community to find new techniques to break the proposed obfuscation method.

***Index Terms*—Functional Locking, Obfuscation, High-Level Synthesis, eFPGA**

I. Introduction

Computer architecture is currently undergoing a dramatic transformation. The breakdown of Dennards scaling implies that power densities are not constant anymore and hence, new architectures are required in order to continue building faster computers within the smallest possible power budget. The main approach so far has been to build heterogeneous architectures, both at the computer level, by having CPUs with GPUs and/or FPGAs, and at the chip level ,by having Systems-on-Chip (SoC) composed of multiple embedded processors, memory, interfaces, and different types of dedicated hardware accelerators. These accelerators are often the main differentiating component of different competing SoC offerings. It is therefore important for companies to protect these IPs from being reversed engineered.

Various techniques have been proposed to address this issue. These include camouflaging [1], split manufacturing [2], and functional locking [3]–[6]. Out of all these techniques, functional locking is the most widely adopted because it is more practical and easier to implement. One of the main problems with traditional functional locking methods, is that they rely on inserting the locking mechanism within the design, which in turn is sent to a potentially untrusted fab to be fabricated. This implies that the fab always has access to the entire design, including the locking circuitry. Moreover, many hardware design companies now focus on developing the overall architecture and often sub-contract the VLSI design back-end (physical design) to a third-party. These third-party companies also have access to the locking mechanism. One promising new approach is to functionally lock a circuit through *omission* [7]. The main idea is to map a portion of the design onto an embedded FPGA (eFPGA). Fig. 1 provides an overview of the main differences between these two locking mechanisms. Fig. 1 (a) shows an example of a traditional approach. In this case, two extra gates are added to the circuit. The key in this case for these two gates would be OR=0 and AND=1. This key is typically stored in an external tamper proof memory as shown in the figure. Fig. 1 (b) shows the new approach where a portion of the design is mapped onto the eFPGA. In this case, the bitstream that configures the eFPGA acts as the logic locking key, and because this bitstream is not made available to third party companies, including the fab, it is harder if not impossible to break. There are multiple commercial eFPGAs that allow to license their eFPGAs as IPs that are inserted as a hardmacro directly into an ASIC design. These include traditional fine-grain eFPGAs [8] and coarse-grain eFPGAs [9].

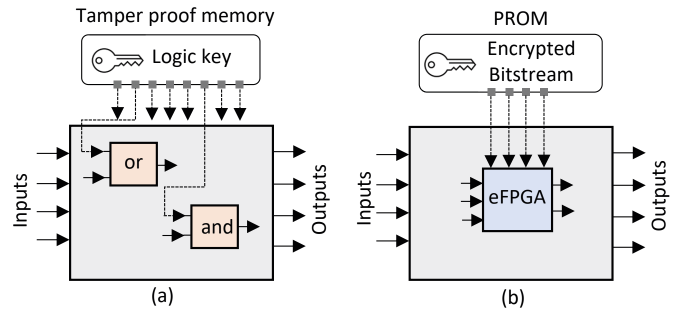

Fig. 1. Logic locking overview. (a) Traditional flow based on logic locking circuitry. (b) Logic locking through omission using an eFPGA.

One advantage of this approach is that FPGA bitstreams are normally encrypted, and hence, can be stored in a regular external memory (PROM). One of the main problems with

978-1-6654-3219-1/21/$31.00 ©2021 IEEE
DOI 10.1109/ICCD53106.2021.00095

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on October 09,2022 at 04:06:31 UTC from IEEE Xplore. Restrictions apply.

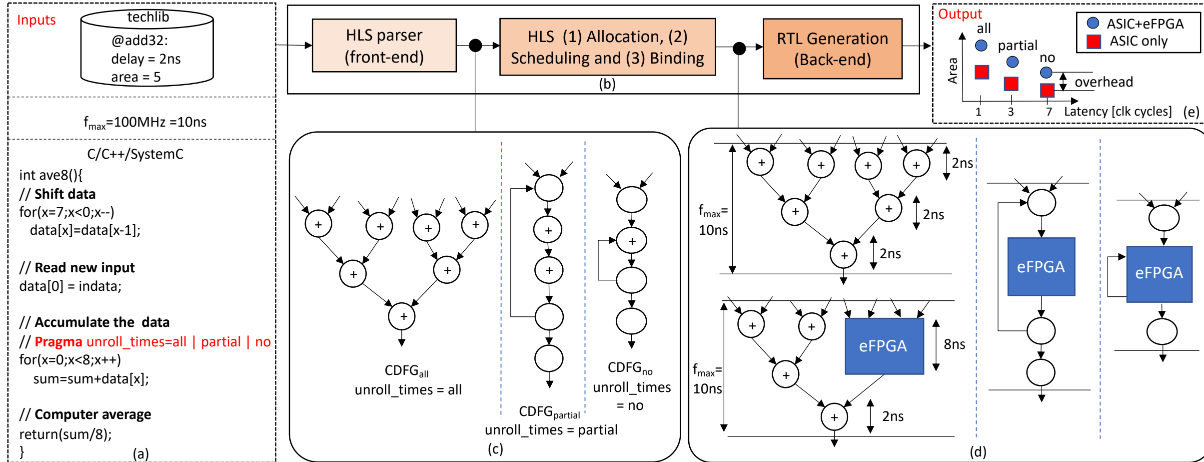

Fig. 2. Motivational Example. (a) High-Level Synthesis inputs. (b) Main High-Level synthesis steps. (c) Results of HLS parser stage (front-end). Different CDFGs are generated based on the synthesis directives annotated at the original C code. (d) Result of HLS scheduling stage when different portions of the CDFG are mapped onto an eFPGA, (e) Final output of the proposed flow showing that the original latencies are preserved, but that the split design (ASIC+eFPGA) lead to circuits with larger area.

this approach is the potential large overheads involved with mapping a portion of the design to the eFPGA. The authors in [10] reported a 10× area, delay and power overheads of FPGAs vs. ASIC designs. It is therefore important to develop a framework that can minimize this overhead. It should nevertheless be noted that for some applications the area overhead is not as important. One example are military applications. Most military agencies and their suppliers are currently forced to use a very limited number *secure* foundries. This makes their ICs extremely expensive. Being able to use cheaper, but less secure foundries overseas, would make their ICs much cheaper, even if the ICs are larger due to the eFPGA embedded in them to make them secure.

Raising the level of abstraction from the RT-level to the behavioral level could help minimizing this overhead. High-Level Synthesis (HLS) takes as input an untimed behavioral description and generates efficient RTL code (Verilog or VHDL). HLS has many advantages over traditional RT-level approaches. It reduces the number of lines of code required, thus, speeding up the design and verification process. It also allows generating different micro-architectures from the same behavioral description with unique area vs. performance and power trade-offs. This is typically done by setting different combinations of synthesis directives in the form or pragmas (comments) at the source code. For example, this allows to control how to synthesize arrays (RAM or registers), loops (unroll, not unroll, partial unroll or pipeline), and functions (inline or not). The HLS process itself can be divided into three main phases: Phase 1 parses the behavioral description and performs technology-independent optimizations such as constant propagation, dead code elimination, and automatic bitwidth reduction. The output of this phase is typically an intermediate representation of the input description in the form of a Control Data Flow Graph (CDFG). Phase 2 performs the three main steps in HLS: resource allocation, scheduling, and binding. Finally, phase 3 generates the resultant circuit in Verilog or VHDL.

The main idea behind this work is to split the CDFG generated after phase 1 ($CDFG_{Cin}$) into the part to be mapped onto the ASIC ($CDFG_{ASIC}$) and the part to be hidden and thus, mapped to the eFPGA ($CDFG_{eFPGA}$) such that $CDFG_{Cin} = CDFG_{ASIC} \cup CDFG_{eFPGA}$. The partitioned design is in turn synthesized (phase 2 of HLS) separately (ASIC and eFPGA) with different constraints such that the generated circuit complies with the original specifications. HLS basically provides a way to re-optimize the partitioned description as opposed to partitioning the design at the RT-Level or gate netlist level where the circuit implementation has already been set and, hence, there are fewer opportunities for re-optimizations. It is therefore tempting to investigate whether the overheads associated with using eFPGAs for logic locking can be minimized through HLS. In summary, the main contributions of this work are:

- We present an automated framework that partitions an untimed behavioral description for HLS into a visible ASIC part, and obfuscated eFPGA part.
- We perform extensive experimental investigation whether the proposed approach works for a variety of CDFGs generated from HLS design space exploration and include a flexible security evaluation platform.

II. Motivational Example

Fig. 2 shows a motivational example of a behavioral description that computes the moving average of eight numbers. Fig. 2(a) shows the other inputs required to synthesize the behavioral description. In particular, a technology library (*techlib*) that contains the area and delay of functional units and a target HLS synthesis frequency (f_{max}). The example also shows the use of synthesis directives in the form of pragmas to force the HLS process to synthesize the circuit in a particular way. In this case, the main loop that accumulates the 8 numbers can be fully unrolled, partially unrolled, or not

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on October 09,2022 at 04:06:31 UTC from IEEE Xplore. Restrictions apply.

unrolled. When parsing the behavioral description, the HLS front-end reads these synthesis directives in and generates a CDFG based on the data dependencies in the code and these pragmas. In this example, three possible CDFGs are shown in Fig. 2(c): $CDFG_{all}$, $CDFG_{partial}$ and $CDFG_{no}$ for the cases that the main loop is fully unrolled, partially unrolled (unroll factor of 2), or not unrolled at all. As mentioned in the introduction, this front-end also does technology-independent optimizations. The main HLS stage then times the CDFG based on the data dependencies, the delay of the different functional units (FUs), and the target synthesis frequency. As shown, in this case, f_{max}=100MHz, which implies a critical path delay of 10ns. This, in turn, translates into the scheduler placing CDFG operations into 10ns clock steps. Because in this example, the delay of the adders=2ns, all the operations in the fully unrolled case can be chained and scheduled within one clock cycle at the output of the main HLS stage (Fig. 2(d).

The main idea in this work is to extract portions of the CDFG and map them onto an eFPGA such that the original schedule is preserved, as shown in the Fig. 2(d). If, e.g., a group of three adders are mapped onto the eFPGA resulting in a delay of 8ns, then the entire CDFG can still be scheduled in a single clock cycle and, hence, the final circuit latency is preserved while meeting the target synthesis frequency.

Moreover, designs of different area vs. latency trade-offs are obtained based on the pragmas specified at the input description as shown. Previous work [11] presented a similar idea but does the partition directly at the behavioral description. This has two unintended consequences: First, the technology-independent optimizations cannot be done when the behavioral description is split into two independent descriptions. Secondly, it does not allow for fine-grained exploration. In the example shown here, only the entire accumulation of the 8 numbers could be mapped to the eFPGA, but not individual FUs as these are not visible at the behavioral description level (a single operation in the C code can lead as to multiple FUs).

Finally, Fig. 2(e) shows the output of the proposed flow, where the red squares show Pareto-optimal designs obtained through specific combinations of pragmas in the behavioral description and the blue circles show the Pareto-optimal designs for the ASIC+eFPGA configuration. It can be observed that in all cases, the latency in clock cycles is preserved as the use of the eFPGA should not lead to additional clock cycles introduced by the obfuscation method. The area between the two solutions is, therefore, the area overhead that the eFPGA-based locking introduces and that this work tries to minimize. Our proposed flow will consequently make use of the slack reported by the HLS process for each scheduled control step, in order to determine which portions of the CDFG should be mapped onto the eFPGA such that the timing is not altered, while guaranteeing that the generated circuit is *secure*. The next section describes our metric for security and the threat model used in this work.

III. Related Work

The protection of intellectual property (IP) is an extremely important topic and has received a lot of attention recently. Even more these days where most hardware design companies are now fabless and have to rely on fabs mostly located offshore to manufacture their ICs. To protect these companies a myriad of techniques have been proposed. These vary from camouflaging [1] to split manufacturing [2] and functional locking [3], [4]. This last technique is seen as the most promising, as it is the least expensive and easiest to implement. The basic idea is to add a locking circuitry to the design such that the circuit does not follow the intended specifications if the logic key is incorrect. This can imply that the outputs are incorrect [3], [4] or that the circuit's performance is degraded [5]. Some functional locking techniques have been specifically proposed at the behavioral level [6]. In this work the authors proposed a method to transform the original behavioral description for HLS to make the generated circuit harder to reverse engineer during chip fabrication, while a key is later inserted to unlock the functionality. Balajandran et al. [12] presented a method that takes into account the scheduling in HLS to insert different types of locking primitives to the behavioral description for HLS. Badier et al. [13] introduced a key-based obfuscation approach to protect BIPs during cloud-based HLS. All of this work makes use of traditional locking mechanisms that insert extra circuitry in the original circuit.

A relatively new idea is to lock by omission. This implies judiciously extracting a portion of the design to be protected and implementing it on an eFPGA [7], [11]. This approach is obviously more secure than the traditional method as the bitstream for the eFPGA now acts as the logic key, and the number of combinations grows with the total number of unique logic functions that can be mapped onto the eFPGA. Moreover, the locking circuitry is not visible to third parties. Recently the authors in [14] used this approach to *hide* the differences between two functionally equivalent accelerators. One obvious problem with this approach is the overhead associated with using FPGAs which we have addressed in this work.

IV. Security Analysis and Metric

This section introduces the threat model assumed in this work and studies the security of our framework. It also introduces a security metric that drives our proposed method. Threat model: The threat model assumed in this work assumes that the attacker has full access to the netlist of the obfuscated design, with the exception of the part mapped onto the eFPGA, and also access to an unlocked, fully working chip that the attacker can interrogate in order to try to find the logic function mapped onto the eFPGA. We also assume that the eFPGA bitstream is encrypted as typically done in commercial FPGAs. Security Analysis: There are two main attacks that could be used to determine the logic function mapped onto the eFPGA: (1) Brute force enumeration or (2) SAT attack. In the first case, the attacker would need to exhaustively enumerate all possible bitstream configurations for the eFPGA until the exact bitstream that replicates the fully-functioning circuit

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on October 09,2022 at 04:06:31 UTC from IEEE Xplore. Restrictions apply.

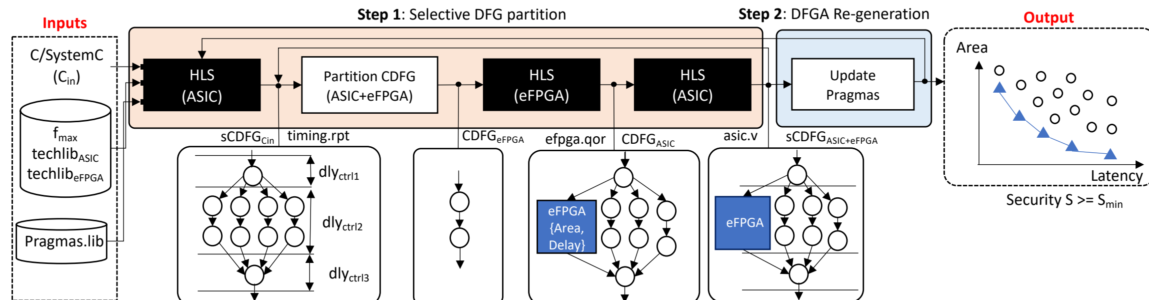

Fig. 3. Overview of proposed flow composed of two steps.

is matched. This implies having access to a fully working chip (available from the market) and using it as an oracle to compare the outputs. This is obviously very time consuming as the bitstream is typically much larger than a traditional locking key as it encodes the functionality of the LUTs and also the interconnect between them. The runtime for a brute force attack would be: $T_{BF} = 2^{bitstream} \times (load_{bitstream} + exec)$, with $bitstream$ being the number of bits that comprise the bitstream, $load_{bistream}$ the time required to load a new bitstream to the eFPGA and $exec$ the time required to run at least one test vector on the newly configured eFPGA which is a function of the latency of the circuit (in clock cycles) and the maximum frequency. Thus, $exec_i = L_i/f_{max}$. It can be observed that the runtime of the attack grows exponentially with the size of the bitstream, leading to extremely long run times for even small bitstream sizes. In the SAT-attack case, the attacker finds the correct key by extracting the Boolean function from the netlist building a miter circuit with two copies of the logic circuit to be obfuscated and finding distinguishing input patterns (DIP) for which two different keys generate different outputs. Keys that lead to incorrect outputs are then discarded until no further DIPs can be found, at which point the SAT-solver returns the correct key. This method has the ability to remove invalid keys at a much faster rate than the brute force enumeration attack and hence, is much faster. Based on this, SAT-attacks cannot be immediately applied to this obfuscation approach as the Boolean function cannot be extracted from the un-programmed eFPGA.

One additional threat to any logic locking mechanism is a removal attack. In this case, the third-party company that has access to the design removes the locking mechanism bypassing it completely. This is obviously not possible here as the circuit will not work correctly without the eFPGA.

Security Metric: Finally, one key parameter required by our proposed method is a security metric to effectively measure how *secure* a partition is. The HLS process reports the area, and performance of the generated circuit, but we also need to guarantee that the generated configuration is secure and discard the ones that are not. Simply using the equation for brute force attack is one option, as it gives an indication of the runtime required to find the logic function mapped onto the eFPGA. The problem with just using this function is that it will tend to accept configurations that only map a single CDFG operation onto the eFPGA, e.g., a single addition or multiplication. In these cases, the attacker could reasonably guess the solution. Thus, in this work we use the bitstream size and the diversity of operations (div_{ope}) mapped to the eFPGA as security metric: $S = bitstream \times div_{ope}$. Intuitively, the larger the variety of logic operations mapped onto the eFPGA, the harder it is to *guess* as oppose to just mapping a single type of operations, e.g., adders or multipliers.

V. Proposed Obfuscation Method

Fig. 3 shows an overview of the complete proposed flow, while algorithm 1 summarizes the first part of our flow. The flow takes as input the behavioral description to be locked (C_{in}) in either ANSI-C or SystemC, the technology libraries for the ASIC ($techlib_{ASIC}$) and eFPGA ($techlib_{eFPGA}$), the target HLS synthesis frequency (f_{max}), a library of pragmas for the behavioral description ($pragma_{lib}$) and the minimum security required (S_{min}). The flow produces RTL code of the Pareto-optimal design configurations with different area (A) vs. latency (L) trade-offs ($PO_i = \{A_i, L_i\}$) where each design meets the specified security metric and where each design is divided into a portion to be mapped onto the ASIC and another onto the eFPGA ($PO_i = RTL_{asic} \cup RTL_{efpga}$). The flow is composed of two main steps. Step 1 generates the CDFG for the given behavioral description and extracts different portions of the CDFG to the eFPGA based on the timing slack available, minimizing the area and delay overheads and meeting S_{min}. It then synthesizes (HLS) the partitioned design targeting only the eFPGA and back-annotating the area and timing result such that the HLS process can re-optimize the synthesis of the ASIC portion. Step 2 sets a different mix of pragmas in the C code in order to generate CDFGs of different nature and thus, generate secure configurations with different trade-offs. Step 1 is repeated for this new CDFG. These steps are described in detail below:

Step 1: Selective CDFG Partition: This first step takes as input the behavioral description to be obfuscated (C_{in}), including a specific set of synthesis directives (pragmas), and synthesizes (HLS) it targeting the ASIC portion only (line 1). The output from the HLS process is the RTL description and different reports that our method requires for the partitioning. In particular, the result of the HLS scheduling stage that indicates which portions of the CDFG have been mapped to individual clock step (scheduled CDFG) $sCDFG_{Cin}$, and the

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on October 09,2022 at 04:06:31 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: Selective Extraction to eFPGA

input : $C_{in}, f_{max}, techlib_{ASIC}, techlib_{eFPGA}$
C_{in}: Original behavioral description to be locked
f_{max}: Target synthesis frequency
$techlib_{ASIC}$: ASIC technology library
$techlib_{eFPGA}$: eFPGA technology library
S_{min}: Minimum acceptable security
output: RTL_{ASIC}, RTL_{eFPGA}
RTL_{ASIC}: RTL of the ASIC portion of the design
RTL_{eFPGA}: RTL of the eFPGA portion of the design

1 $(sCDFG_{Cin}, time.rpt) = HLS(C_{in}, techlib_{ASIC}, f_{max})$;
2 $CSList = sort_slack(sCDFG, time.rpt)$;
3 $OPList = sort_num_ope(sCDFG)$;
4 **for** $(cs_i \in CSList)$ **do**
5   **for** $(op_i \in cs_i)$ **do**
6     $cluster_i \leftarrow cluster_nodes(op_i)$;
7     $dly_cluster_i = est_dly(cluster_i, techlib_{eFPGA})$;
8     $S_cluster_i = est_sec(cluster_i, techlib_{eFPGA})$;
9   **if** $(dly_cluster_i \leq cs_i(slack))$ **then**
10   **if** $(S_{min} \leq S_cluster_i)$ **then**
11         $ListCluster[x++] \leftarrow cluster_i$;
12   **end**
13   **end**
14   **end**
15 **end**
16 **for** $(cluster_i \in ListCluster)$ **do**
17   $(A_i, Dly_i) = HLS(cluster_i, techlib_{eFPGA}, f_{max})$;
18   $ASIC_i = anotate_asic(A_i, Dly_i)$;
19   $S_i(A_i, Dly_i, L_i) = HLS(C_{in}, techlib_{ASIC}, f_{max})$;
20   $SList[x++] \leftarrow S_i$;
21 **end**
22 $SList_{sorted} = sort_solutions(SList)$;
23 $S_{smallest} = return_smallest_overhead(SList_{sorted})$;
24 **return** $(S_{smallest} = \{RTL_{ASIC} + RTL_{eFPGA}\})$

timing report (timing.rpt) indicating the delay of each control step and the available timing slack with respect to f_{max}.

The pragmas are important, as shown in the motivational example, because, based on the mix of distinct pragmas, a different CDFG is generated.

This step then continues by parsing the generated scheduled CDFG ($sCDFG_{Cin}$) and sorting the individual control step based on slack available from largest to smallest. $CSList = \{cs_1, cs_2, \ldots, cs_n\}$ with $t_{slack}(cs_1) > t_{slack}(cs_2) > t_{slack}(cs_n)$ (line 2).

The method also traverses the scheduled CDFG ($sCDFG$) and records the number of times that individual operations appear on different control steps. (line 3). The main reason for this is that HLS is very efficient doing resource sharing. In resource sharing, a single functional unit (FU) is shared across different operations in the CDFG. This typically leads to smaller circuits. The main problem with mapping shared operations to the eFPGA is that we have observed that it leads to timing issues, as different control step have different amounts of slack. From the security point of view, resource sharing might actually be beneficial as it allows to map multiple operations to the eFPGA with a minimum area overhead. The result is $OPList = \{op_1, op_2, \ldots, op_n\}$ with $Num(op_1) < Num(op_2) < Num(op_n)$.

The method proceeds by clustering in each control step (cs_i) operations that have a security metric S_i larger than S_{min} and delays smaller than the available slack in that control step (lines 4 to 15). All the clusters that meet the constraints are added to a cluster list (line 11). Although exhaustive enumerations might lead to a large number of combinations, the number of CDFG nodes in a single control step is typically small. It should be noted that the nodes in the cs_i do not need to be directly connected. The delay of every cluster is then estimated by extracting the delay of each operation mapped onto the eFPGA from the FPGA technology library passed as input to the flow ($techlib_{eFPGA}$) (line 7). The method uses the area in terms of LUTs used for that operation as an approximation of bitstream size from the FPGA technology library. This process is repeated for each control step.

Next, all the valid solutions are synthesized (HLS) to get the actual area and delay information for the operations mapped onto the eFPGA only (line 17). The results in terms of area and delay are back-annotated to the rest of the circuit to be synthesized on the ASIC (lines 18 and 19) . The HLS process can then re-optimize the partitioned circuit, typically leading to the same schedule as the original ASIC-only design. Finally, all the results are sorted based on the ability to meet the original schedule, and within those solutions, the one with the smallest area overhead is selected (lines 22 and 23). It should be noted that our proposed method is independent of the target HLS frequency (f_{max}). One could think that lower frequencies increase the amount of logic that can be mapped onto a single control step. Nevertheless, the nature of the HLS process automatically accounts for this and adjusts the amount of logic mapped onto each control step based on f_{max}.

Step 2: Pragma Update: This second step is optional and automatically updates the mix of pragmas specified in the behavioral description. As shown in the motivational example, different pragma mixes lead to a completely new CDFG with unique area vs. performance trade-offs. The main idea behind this step is to fully characterize the behavioral description in terms of area vs. performance when a portion is mapped to the eFPGA. It also helps to investigate whether the overhead of our proposed method increases or decreases with different CDFG structures. If the user knows exactly the timing required for the circuit and fixes the pragmas in the behavioral description, then this phase is not required.

The problem of automatic pragma settings has been widely studied in the area of HLS design space exploration (DSE). A recent survey summarizes the main contributions in this domain [15]. In this work, we make use of recent observations presented in [16], [17] that make use of transfer learning from previous HLS DSE results onto a new unseen description to be explored. Basically, it was shown that behavioral descriptions with *similar* structure have identical pragma combinations that lead to the same Pareto-optimal designs. Thus, the proposed pragma update phase parses the description (C_{in}) and builds an Abstract Syntax Tree (AST) from its dependencies. It extracts all the loops within the description and computes the *similarity* index with other designs that have been pre-characterized in a database. This index is based on perceptual hashing for the AST. Perceptual hashing is an algorithm commonly used in digital forensics. The key advantage of perceptual hashing is

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on October 09,2022 at 04:06:31 UTC from IEEE Xplore. Restrictions apply.

that it is robust enough to take into account dissimilarities but flexible enough to distinguish between different micro-kernel structures. The database has already been generated (omitted for blind review).

The method then goes to the database and finds the AST that has the smallest Hamming distance and extracts the pragma combinations used to generate the Pareto-optimal designs for that design. This method is extremely fast as the entire process does not require any actual synthesis.

Step 1 is repeated for every pragma combination that leads to Pareto-optimal design.

TABLE I
EXPERIMENTAL SETUP

HLS Tool	NEC CyberWorkBench 6.1
HLS Frequency	100MHz
ASIC Logic Synthesis	Synopsys Design Compiler 2018.06-SP1
FPGA Logic Synthesis	Intel Quartus Prime Pro. 21.1
RTL Simulator	Synopsys VCS 0-2018.06
Synthesis Technology	Nangate 45nm Opensource
FPGA Technology	Intel Stratix V

VI. EXPERIMENTAL RESULTS

Table I shows an overview of the experimental setup used to test our proposed flow. The HLS tool used is NEC's CyberWorkBench v. 6.1 and the target synthesis frequency (f_{max}) is set to 100MHz in all cases (10ns control step delay). The target ASIC technology is Nangate OpenCell 45nm, and we use Intel Stratix V as a proxy for an embedded FPGA. The final ASIC partitions is synthesized using Synopsys Design Compiler 2018.06-SP1 and the FPGA partition with Intel Quartus Prime Pro 21.1 to get accurate area results. The work in [18] is used to convert the FPGA logic resources into μm^2. This allows us to compare the area overhead directly. We also perform a functional RTL simulation of the partitioned design and compared it with the ASIC only one to make sure that the partitioned circuits is functionally correct. Synopsys VCS 0-2018.06 is used as RTL simulator. The security metric S was set such that the time to find the correct bitstream would take at least 1 year using a state-of-the-art computer to find the correct bitstream using an exhaustive search and forcing all benchmarks to at least map two CDFG operations on the eFPGA.

Eight different benchmarks from the open-source Synthesizable SystemC benchmark suite (S2Cbench) [19] were used to test our proposed methodology. The benchmarks selected are from different domain and different complexities. Table II summarizes their main characteristics in terms of their domain, lines of code, number of adders and multipliers when the loops are fully unrolled, number of arrays and finally the number of loops. The benchmarks have been grouped based on their complexity into small, medium and large. This should help fully characterize our proposed design obfuscation method.

Two set of different experimental results are presented. The first experiments measure the overhead introduced by our method compared to the ASIC only design as well as the state of the art. The second set of experiments analyze the robustness of our proposed method to reverse engineering.

TABLE II
BENCHMARK DETAILS

	Bench	Domain	Lines	Mul	Add	Arrays	Loops
Small	ave16	DSP	24	0	16	1	1
	fir	DSP	105	9	9	2	1
Med	interp	DSP	212	8	21	1	5
	kasumi	encrypt	286	0	22	16	8
	snow3G	encrypt	372	9	36	3	4
Large	decim	DSP	422	5	50	10	16
	jpeg	image	480	121	98	11	32
	cnn	image	331	15	446	16	51

Overhead Analysis: Table III compares our proposed method vs. an ASIC only implementation (ASIC) and an ASIC+eFPGA based on [11] that does the partitioning at the source code level when all the loops are unrolled and all the functions inlined in order to find the design with the highest performance (lowest latency). The results show that our proposed method does never change the original latency of the circuit. Although the delay increases in most cases, it only does on average by 14% while always being able to meet the target synthesis frequency of 100MHz. This is a significant improvement over the state-of-the-art, which in all cases increased the latency by an average of $2\times$ with respect to the original ASIC-only design.

Our proposed method also leads to a much smaller area overhead of an average of $1.70\times$, while the previous work leads to circuits on average $5.59\times$ larger. That is a $3.29\times$ area improvement while keeping the latency equal to the original circuit.

Fig. 4 shows the results when the Pareto-optimal designs for each of the benchmarks are obfuscated through our proposed method. Every point in the curve represents a different CDFG generated through different combinations of synthesis options. It can be observed that in all cases, our proposed method guarantees to keep the original latency constant while leading to a much smaller area overhead than previous work. On average, across all of the benchmarks and all of their different implementations, our proposed method leads to an area overhead of $1.57\times$, while previous work leads to an average area overhead of $8.74\times$. Moreover, it is important to observe that our proposed method guarantees in all cases that the original latency is preserved.

Finally, Table IV compares the two obfuscation methods in terms of their runtime to obtain the results depicted in Table III. Our proposed method is on average $1.82\times$ slower due to the larger search space that needs to be evaluated. The running time is nevertheless still very small, e.g., it takes at most 11 minutes for the cnn benchmark. We believe that this is still acceptable, considering the benefits introduced by our flow.

In summary, based on these results, we can claim that our proposed method is more efficient than previous work, leading to smaller overheads while maintaining the original circuits' timing.

Security Analysis: The next set of results investigate the robustness of our proposed method to being reverse engineered. The assumptions that we make here is that the rough designer

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on October 09,2022 at 04:06:31 UTC from IEEE Xplore. Restrictions apply.

TABLE III
COMPARISON BETWEEN THE PROPOSED METHOD (ASIC+EFTPGA), ASIC ONLY IMPLEMENTATION AND THE OBFUSCATION METHOD IN [11] (ALL LOOPS UNROLLED AND ALL FUNCTIONS INLINED).

Benchmark		ASIC			ASIC+eFPGA [11]			ASIC+eFPGA (Proposed)		
		Area [μm^2]	Delay [ns]	Latency [clk cycles]	Area [μm^2]	Delay [ns]	Latency [clk cycles]	Area [μm^2]	Delay [ns]	Latency [clk cycles]
Small	ave16	805	2.42	1	37,919	2.57	2	11,847	3.47	1
	fir	5,375	3.14	1	85,341	6.85	2	6,740	8.04	1
Medium	interp	15,610	7.02	1	148,311	9.09	2	17,615	7.48	1
	kasumi	70,566	4.41	2	107,006	4.41	9	79,861	4.41	2
	snow3G	12,294	1.31	10	63,861	1.68	41	21,390	1.31	10
Large	decfilt	25,001	9.95	5	81,486	9.97	6	41,540	9.95	5
	jpeg	76,439	9.45	450	151,231	9.78	456	76,786	9.97	450
	cnn	181,950	9.12	33	481,950	9.54	37	186,950	9.957	33
Average			**5.9**			**7.2**			**6.8**	
Geomean		**19,437**		**5.9**	**108,691**		**11.5**	**33,077**		**5.9**

TABLE IV
RUNTIME COMPARISON ASIC+EFPGA [11] VS. PROPOSED METHOD

Bench	ASIC+eFPGA [s] ([11])	ASIC+eFPGA [s] (Proposed)	Diff
ave16	37	53	1.43
fir	21	58	2.76
interp	96	157	1.64
kasumi	191	423	2.21
snow3G	263	325	1.24
decfilt	113	248	2.19
jpeg	350	454	1.30
cnn	379	676	1.78
Avg.			**1.82**

that wants to reverse engineer the design can lawfully purchase the IC from the market and that he can fully reverse engineer the ASIC portions of the IC as well as identify the eFPGA part identifying the number of LUTs and switch boxes. This has been shown possible through different methods including electromagnetic imaging [20], [21]. We also assume that the bitstream is fully encrypted and thus, cannot be reversed engineered, but that the attacker is able to generate valid bitstreams to program the eFPGA, although this by itself is not trivial.

Fig. 5 shows an overview of the security evaluation platform that we created to simulate how the attacker could break our proposed obfuscation scheme. For this, we synthesize the ASIC portions of the design using Synopsys Design Compiler. For the FPGA portion we create a model of the eFPGA using VTR [22] and *program* that model through a bitstream generator. This is done by describing the target eFPGA architecture using VTR's XML-based architecture description replicating the Intel Stratix V architecture used in this work as proxy for an eFPGA. We then synthesize the RTL portion of the design to be mapped onto the eFPGA using ABC, followed by the technology mapping of the netlist and placing and routing the netlist on the eFPGA fabric to obtain the post-implementation netlist. A simple bitstream generator is created for the placed and routed netlist.

This netlist is integrated with the ASIC portion netlist. We also build, as shown in Fig. 5, a testbench that takes these netlists as DUT. This testbench takes a minimal subset of test vectors (TVs) as inputs. Here we used only 10 as this is enough to discard if the eFPGA is configured incorrectly. The testbench also contains the golden outputs for those 10 test vectors. The entire system can then be simulated and the generated outputs are automatically compared with the golden outputs. A *match* signal is generated if the functionality is correct, signaling that the eFPGA was programmed correctly and *no match* if it is not.

This platform is used to test the robustness of our proposed flow. As shown in Fig. 5 we also developed a bitstream generator. This bitstream generator knows the eFPGA structure and the bitstream format (taken from the correct bitstream generated from the VTR flow) and generates new bitstreams that in turn correspond to new post-implementation netlists, and hence, a new eFPGA configuration. The testbench feeds the simulation result back to the bitstream generator to alert the attacker if the generated bitstream is correct or not, who can then continue generating new bitstreams until a valid one is found.

We evaluated this flow for the eight benchmarks and left it running for one week each time without success of replicating the correct bitstream that successfully configured the eFPGA. The bitstream generator is based on a pseudo-random generation approach that generates random bitstream configurations, but only considering valid bitstreams. Many bitstreams (e.g., all 0's or all 1's or bitstreams that lead to short circuits) are discarded, thus, reducing the search space. The total number of combinations of valid bitstreams are set in our proposed partitioning method to at least 500 million. This is the equivalent to finding the correct bitstream in one year considering that each evaluation would takes 100ms.

One of the obvious problems with this platform is that it is simulation based, which makes it relatively slow. We measured average simulation times of 1s to 3 seconds for benchmarks of different complexities for a single evaluation. In reality the attacker would have access to the real hardware which would allow him to theoretically evaluate each new bitstream faster than our simulated approach. We say *theoretically* because every newly generated bitstream would need to be encrypted using the public key available to the attacker and the bitstream would need to be scanned into the eFPGA before the functionality can be evaluated. This is also relatively time consuming considering that the bitstream now also has to be decrypted and typically also decompressed at the eFPGA side.

VII. ACKNOWLEDGMENTS

This work is partially supported by the NSF Industry/University Cooperative Research Center on Hardware and

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on October 09,2022 at 04:06:31 UTC from IEEE Xplore. Restrictions apply.

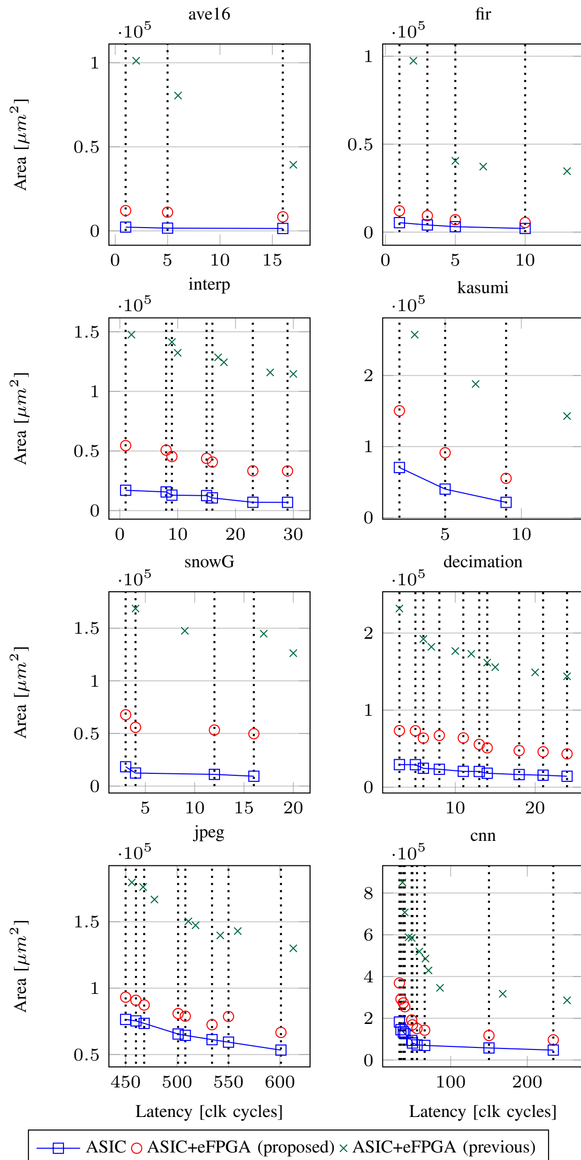

Fig. 4. Obfuscation results for different versions of the same application comparing ASIC only baseline vs. our proposed and previous work

Embedded Systems Security and Trust (CHEST) through project #P7_20.

VIII. Conclusion

This work has presented a method to efficiently obfuscate a circuit generated from High-Level Synthesis by judiciously extracting a small portion of it onto an eFPGA. Making use of the detailed timing information reported by the HLS process allows our proposed method to maintain the original timing. Our proposed method is extended to generate a set of Pareto-optimal configuration by setting different mixes of synthesis directives in the form of pragmas. Experimental results show that our proposed method works well and that it is a competitive alternative to traditional functional locking methods while being more secure because the eFPGA bitstream now acts as a locking key.

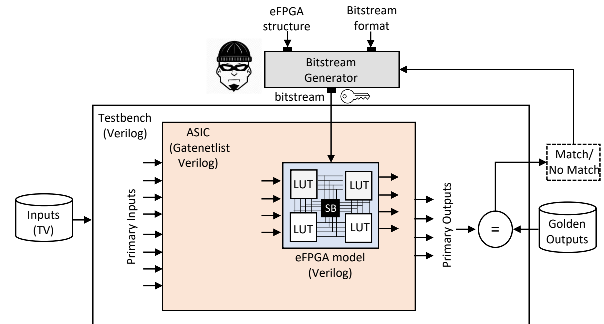

Fig. 5. Overview of system developed to find correct eFPGA bitstream.

References

[1] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran, "CamoPerturb: Secure IC camouflaging for minterm protection," in *ICCAD*, 2016, pp. 1–8.

[2] J. Rajendran, O. Sinanoglu, and R. Karri, "Is split manufacturing secure?" in *DATE*, 2013, pp. 1259–1264.

[3] J. A. Roy *et al.*, "EPIC: Ending Piracy of Integrated Circuits," in *DATE*, 2008, pp. 1069–1074.

[4] 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.

[5] M. Yasin *et al.*, "What to Lock? Functional and Parametric Locking," in *GLSVLSI*. ACM, 2017, p. 351–356.

[6] C. Pilato *et al.*, "TAO: Techniques for Algorithm-Level Obfuscation during High-Level Synthesis," in *DAC*, 2018, pp. 1–6.

[7] M. M. Shihab *et al.*, "Design obfuscation through selective post-fabrication transistor-level programming," in *DATE*, 2019, pp. 528–533.

[8] Quicklogic, "ArticPro, https://www.quicklogic.com," 2021.

[9] Renesas, "STP, http://www.renesas.com/products/soc/asic/programmable," 2021.

[10] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE TCAD*, vol. 26, no. 2, pp. 203–215, 2007.

[11] B. Hu, J. Tian, M. Shihab, G. R. Reddy, W. Swartz, Y. Makris, B. Carrion Schaefer, and C. Sechen, "Functional Obfuscation of Hardware Accelerators through Selective Partial Design Extraction onto an Embedded FPGA," in *GLSVLSI*, 2019, p. 171–176.

[12] A. Balachandran and Benjamin Carrion Schafer, "Efficient Functional Locking of Behavioral IPs," in *MWSCAS*, 2020, pp. 639–642.

[13] H. Badier, J.-C. L. Lann, P. Coussy, and G. Gogniat, "Transient Key-based Obfuscation for HLS in an Untrusted Cloud Environment," in *DATE*, 2019, pp. 1118–1123.

[14] J. Chen *et al.*, "DECOY: DEflection-Driven HLS-Based Computation Partitioning for Obfuscating Intellectual PropertY," in *DAC*, 2020, pp. 1–6.

[15] B. Carrion Schafer and Z. Wang, "High-Level Synthesis Design Space Exploration: Past, Present, and Future," *IEEE TCAD*, vol. 39, no. 10, pp. 2628–2639, 2020.

[16] Z. Wang *et al.*, "Efficient and robust high-level synthesis design space exploration through offline micro-kernels pre-characterization," in *DATE*, 2020, pp. 145–150.

[17] L. Ferreti *et al.*, "Leveraging prior knowledge for effective design-space exploration in high-level synthesis." in *ESweek*, 2020, pp. 1–8.

[18] H. Wong, V. Betz, and J. Rose, "Comparing FPGA vs. Custom CMOS and the Impact on Processor Microarchitecture," in *FPGA*, 2011, p. 5–14.

[19] B. Carrion Schafer and A. Mahapatra, "S2CBench: Synthesizable SystemC Benchmark Suite for High-Level Synthesis," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 53–56, Sept 2014.

[20] R. Omarouayache and P. Maurine, "An Electromagnetic imaging technique for Reverse Engineering of Integrated Circuits," in *APACE*, 2016, pp. 352–357.

[21] B. Lippmann, M. Werner *et al.*, "Integrated flow for reverse engineering of nanoscale technologies," in *ASP-DAC*, ser. ASPDAC '19, 2019, p. 82–89.

[22] J. Rose *et al.*, "The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing," in *FPGA*, 2012, p. 77–86.

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on October 09,2022 at 04:06:31 UTC from IEEE Xplore. Restrictions apply.