Functional Obfuscation of Hardware Accelerators through Selective Partial Design Extraction onto an Embedded FPGA

Bo Hu, Jingxiang Tian, Mustafa Shihab, Gaurav Rajavendra Reddy, William Swartz,
Yiorgos Makris, Benjamin Carrion Schaefer, Carl Sechen
{bo.hu,jxt122130,mustafa.shihab,gaurav.reddy,bill-swartz,yiorgos.makris,schaferb,carl.sechen}@utdallas.edu
Department of Electrical Computer Engineering, The University of Texas at Dallas,Richardson, TX, U.S.A, 75080-3021

ABSTRACT

The protection of Intellectual Property (IP) has emerged as one of the most serious areas of concern in the semiconductor industry. To address this issue, we present a method and architecture to map selective portions of a design, given as a behavioral description for High-Level Synthesis (HLS) to a high-security embedded Field-Programmable Gate Array (eFPGA). In this manner, only the end-user has access to the full functionality of the chip. Using six benchmark circuits, we show that our approach is effective. In all cases, the Time-To-Break (TTB) is so long (at least 8 million hours) that for all practical purposes the designs are secure, while incurring area overheads of around 5%. Further, latencies were only slightly increased, while the computation times are under one minute.

CCS CONCEPTS

• **Security and privacy** → **Security in hardware**; **Hardware-based security protocols**;

KEYWORDS

hardware security; obfuscation; high level synthesis; embedded FPGA

ACM Reference format:
Bo Hu, Jingxiang Tian, Mustafa Shihab, Gaurav Rajavendra Reddy, William Swartz, and Yiorgos Makris, Benjamin Carrion Schaefer, Carl Sechen. 2019. Functional Obfuscation of Hardware Accelerators through Selective Partial Design Extraction onto an Embedded FPGA. In *Proceedings of Great Lakes Symposium on VLSI 2019, Tysons Corner, VA, USA, May 9–11, 2019 (GLSVLSI '19),* 6 pages. ACM, New York, NY, USA.
https://doi.org/10.1145/3299874.3317992

1 INTRODUCTION

Contemporary semiconductor manufacturing largely follows a fabless business model wherein third-party foundries are provided with the source files (i.e., in GDSII format) of an integrated circuit (IC) design and are contracted to fabricate it. Considering the geopolitical position of the majority of the semiconductor manufacturing industry, this fabless model incurs an inherent security and trustworthiness risk. More specifically, the entire intellectual property (IP) of an IC design is exposed to the-potentially untrusted-foundry or any rogue element therein and is, therefore, subject to malicious manipulation and/or theft. As a result, protecting sensitive parts of the design (*e.g.*, trade secrets, classified data/algorithms, competitive advantage circuits, etc.) and ensuring functional integrity of the received ICs becomes very challenging. Furthermore, such security and trust concerns continue to exist after an IC is deployed in the field of operation. Indeed, reverse engineering may still reveal the secret IP contained in an IC, while dormant malicious logic can be activated post-deployment in order to compromise its functional integrity. Accordingly, design obfuscation and hardware Trojan solutions are urgently required.

These fundamental development and supply chain changes have led to serious security and trustworthy concerns, as these companies now do not have control over the manufacturing process and hence do not control how many ICs the foundry has actually manufactured. To address this issue we propose to extract selectively parts of an ASIC design and map it into an embedded FPGA (eFPGA) with the objective of obfuscating the functionality of the design.

Intelligent attacks, such as the use of a Boolean Satisfiability (SAT) solver [14], have been developed and compromised most early obfuscation schemes. This, has resulted in an on-going race between new defenses and new attacks, reinforcing the need for further research in this area [4, 5, 16]. In this work, we utilize a design obfuscation approach that omits parts of it from the fabricated silicon and reinstates it via post-manufacturing programming. We will show that using FPGA-style lookup tables (LUTs) to implement the omitted portion of the function provides high security levels.

The main contribution of this work is an algorithm to extract a portion of a design to be mapped onto the eFPGA, such that there is a limited area and delay overhead while providing a very high security benefit.

2 RELATED WORK

The obfuscation based approach transforms a given circuit into a functionally equivalent circuit that is significantly more difficult (ideally impossible) to reverse engineer. Some examples include the dedicated obfuscation of DSP circuits by performing high-level transformations during the design stage and inserting multiplexers in the datapath controlled by a FSM [4].

Due to the significance of the targeted problem, the last decade has seen a flurry of research activity in a number of different directions. Concepts such as Active Metering [2, 11], Logic Encryption [10, 17], State Obfuscation [3], Design Camouflaging [9], Split Manufacturing [8], and many others have been introduced. Each such solution comes with each own strengths and limitations and, to date, no single solution has been successful in addressing the security

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
GLSVLSI '19, May 9–11, 2019, Tysons Corner, VA, USA
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6252-8/19/05.
https://doi.org/10.1145/3299874.3317992

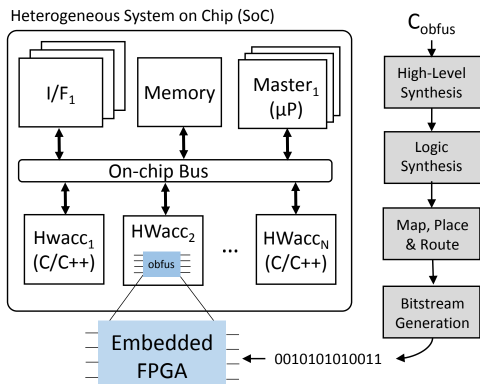

Figure 1: Design obfuscation via post-fabrication programming of an embedded FPGA (eFPGA) of a portion of a dedicated hardware accelerator.

and trust challenges in a cost-effective manner. Split manufacturing, for example, divides the design into front end of line (FEOL) and back end of line (BEOL) and uses different foundries for each part, so that the entire design is never exposed. Yet two foundries are required, along with additional logistics for completing the design. Design time and performance overhead is also potentially incurred. Similarly, design camouflaging through the use of library cells that contain decoy elements can make reverse engineering far more difficult (yet still not impossible) but it also incurs significant overhead and does not protect the GDSII file from an attacker at the foundry. Logic encryption and state obfuscation embed a design within a larger space, wherein only a correct key will yield the correct functionality. These solutions are very well studied and understood, yet when applied at scale they do incur significant overhead. Furthermore, an unlocked IC is still subject to reverse engineering, unless additional costly provisions are taken in order to hide the key or individualize it per fabricated IC.

Closer to our work, the authors in [14] proposed to implement dedicated functions using reconfigurable logic to obfuscate the design and show an example of obfuscating the instruction decode unit of a LEON2 open source SPARC V8 architecture processor. Their approach is nevertheless ad-hoc. In [16] the authors present a method to extract at the gate-netlist level different paths and map their gates to non-volatile spin transfer torque (STT) based reconfigurable LUTs. Although a promising approach, it is currently not feasible to fabricate these types of hybrid circuits. The authors in [13] also extract at the gate-netlist level and map the obfuscation part to the transistors in a fine-grained embedded FPGA.

Our work is different from the previous work in that we propose a design flow for mapping different design portions at the behavioral level and hence can understand the security vs. overhead trade-offs early on in the design flow.

3 EMBEDDED FPGA

The obfuscated portions of a system design, determined by the algorithms presented in this work, can be mapped to any embedded FPGA (eFPGA). There are multiple commercial eFPGAs on the market. The most notable vendors include Achronix [1] and Quicklogic [7]. These eFPGAs are based on traditional 4-input, 1-output LUTs and can contain multiple hard macros such as embedded DSP modules and memories.

The key issue is then how to find an ideal portion of a design, given as behavioral decription for high-level synthesis (HLS), to map onto an eFPGA [15] to enable effective design obfuscation.

Problem Formulation: Given a behavioral description C to be synthesized as an ASIC, selectively extract different portions of C to be mapped onto the eFPGA, such that $C = C_{ASIC} \cup C_{eFPGA}$, and such that by extracting different C_{eFPGA}, a unique list of obfuscated designs (D_{obfusi}) are obtained, $D_{obfus}List = \{D_{obfus1}, D_{obfus2}, \ldots, D_{obfusn}\}$, with unique area, latency and security trade-offs, with $D_{obfusi} = \{A_i, L_i, S_i\}$, where A_i is the area, L_i the latency and S_i the security metric. Out of all $D_{obfus}List$, we are only interested in the pareto-optimal ($D_{obfus}(opt)$), where a Pareto-optimal design can be defined as a design in which it is impossible to improve one metric without making at least one of the other metrics worse.

The fundamental concept behind using an eFPGA for design obfuscation is shown in Fig. 1. Specifically, we propose to replace the sensitive parts of a design, which we wish to withhold from an untrusted foundry, with the eFPGA. After the chips are fabricated and received from the foundry, the withheld design portions are programmed into the eFPGA fabric to complete the IC functionality. This idea holds great potential in addressing both IP protection and design integrity concerns.

Without the programming bits in place, the functionality of the eFPGA cannot be determined. For example, when the layout of a design is in the untrusted foundry or when a chip is not programmed, the eFPGA block has no meaningful functionality and the required program cannot be extracted, as it does not exist. This work assumes that the bitstream is encrypted, similar to commercial FPGAs and hence cannot be reverse engineered nor copied.

4 SECURITY METRIC

In order to successfully map different portions of a circuit to the eFPGA, we need a security metric that quickly and effectively reveals how secure the resultant circuit is, in addition to the area and performance overheads associated with it. Approximate area and performance overheads can be obtained from the HLS tool once C_{ASIC} and C_{eFPGA} are synthesized into RTL. Thus, a security metric (S) is required such that after HLS our proposed method can determine the security level of each configuration.

For example, if only a single-line of C code is mapped to an eFPGA block, such as a large multiplier, an attacker might reasonably try to guess that a single operator was being obfuscated. Therefore, in the context of HLS, a key aspect of obfuscation is the number of operators being assigned to an eFPGA block. Another important factor is the number of cells to be implemented in the eFPGA, as larger numbers of cells greatly impairs any possible SAT-based (or brute-force) attack. A SAT-based attack must enumerate all possible cells (each "cell" being implemented by one LUT) in the eFPGA. Thus, we use the product of the number of cells ($cell_i$) and the number of operators (op_i) as the security cost function (S_i) for each obfuscated design (D_{obfusi}):

$$S_i = cell_i \times op_i \tag{1}$$

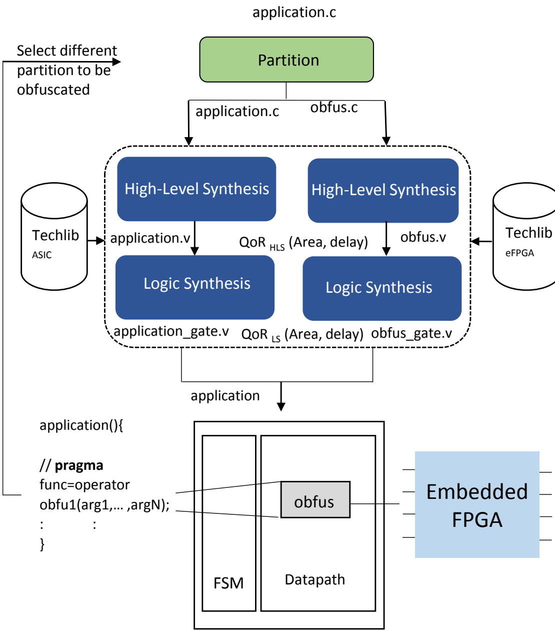

Figure 2: Design obfuscation design methodology starting from a behavioral description for HLS.

After we obtain the pareto-optimal designs ($D_{obfus}(opt)$) using the security metric in Eq. 1, we will select one of those designs with an acceptable area overhead and compute the time-to-break (TTB) for both a SAT-based and brute-force based attack. The TTB for a SAT-based attack is given by:

$$TTB_{SBA} = 100^{cell_i} \times max(bitstreamload_i, executetime_i) \tag{2}$$

Here we conservatively assume that each LUT can only implement 100 different logic functions. We further conservatively ignore the substantial obfuscation due to the unknown (to the attacker) switch-box routing. $Bistreamload_i$ is the time it takes to load the new bitstream to the eFPGA, and $executetime_i$, the execution time of a single test vector. Here, $bitstreamload_i = Bitstream_i \times clockperiod$ and $executetime_i = latency_i \times clockperiod$, where the latency is in clock cycles and the clock period is the inverse of the operating frequency.

Besides a SAT-based attack, a brute-force attack is also possible. We assume that the attacker has knowledge of the eFPGA, and knows how to generate a bitstream but does not know which bitstream configurations lead to a valid circuit (a set of configured logic gates and their interconnections). In this type of attack, the following steps are iteratively applied: 1) a bitstream is generated, 2) the bitstream is loaded into the eFPGA block and 3) at least one test vector is executed to determine if the configuration is valid or not. This test is indispensable but not sufficient. In the case that a valid output is generated, then other test vectors would need to be applied to fully guarantee the correctness of the complete circuit. Assuming one circuit portion mapped to an eFPGA obtained from C_eFPGA, requiring a $bitstream$ to be configured:

$$TTB_{BFA} = 2^{bitstream_i} \times max(bitstreamload_i, executetime_i) \tag{3}$$

ALGORITHM 1: Function encapsulation

```
C_obfus_single: New generated Behavioral description
sub_module: obfuscated module to be mapped to eFPGA
top_module: top module to be mapped to ASIC
    input:  C_orig Behavioral description
1  /* Code Annotation : annotate by pragma */
2  while (line ∈ C_orig) do
3  |   annotate by /* pragma forloop */ or
4  |   /* pragma if-else */ or
5  |   /* pragma operation line */ ;
6  end
7  /* FuncEncp: Function encapsulation */
8  for ( each pragma ∈C_orig) do
9  |   generate sub_module;
10 |   generate top_module;
11 end
   output:  set of pairs {C_obfus_single} = { sub_module,
               top_module}
```

where TTB_{BFA} is a function of the bitstream size ($bitstream$), which is in turn a function of C_{eFPGA}, since the larger the portion of the application mapped to the eFPGA, the larger the bitstream becomes.

5 OBFUSCATION DESIGN METHODOLOGY

This section describes the methodology used to find which portion or portions of a behavioral description for HLS should be mapped to an eFPGA block, trading off a limited amount of area/power overhead for a high level of security. The proposed obfuscation method starts with a library pre-characterization step for the ASIC and eFPGA, based on the technology that the design team is targeting. This is important to get accurate area and delay information right after HLS. The results of this pre-characterization step is a technology library, which is in turn used by our selective extraction method. It then proceeds, following a divide and conquer strategy, by selecting individual sections of the behavioral descriptions and mapping them onto the eFPGA block. Finally, it merges all the results to obtain the best solutions. The next subsections describe these main steps in detail, as explained in Algorithm 2.

5.1 Pre-characterization and Library Generate

HLS can be defined as the process of converting an untimed behavioral description into a Register Transfer Level (RTL) description that can efficiently execute it. HLS executes three main steps: Resource allocation, scheduling and binding. In the scheduling step, the synthesizer schedules the different operations in the code based on the data dependencies and delay information of each operator.

Thus, in order for the synthesizer to successfully schedule operations in a control step, the synthesizer needs to know the delay of each functional unit (FU). For this purpose, commercial HLS tools provide library characterizers which extract the area and delay information of every basic primitive for the target technology/eFPGA.

Therefore, the technology library of the ASIC and the eFPGA block are generated in this pre-characterization stage. This allows our method to use a commercial HLS tool and get accurate preliminary results of how mapping different portions of the code onto

the eFPGA affects the overall area and performance. We call this QoR_{HLS} (Quality of results).

This step only needs to be executed once, unless the target technology changes. Once the technology libraries for the ASIC and eFPGA are generated (lines 2-3 in Algorithm 2), our proposed method can continue by selectively extracting different portions of the behavioral description and quickly quantifying the effect of this extraction on the area, delay and security of the circuit.

5.2 Selective extraction

Selecting the portion to obfuscate at the behavioral level has the advantage of allowing the design team to know exactly which portion of the design is obfuscated, whereas this is not straightforward at the gate-netlist, where the design team might lose intuition on what exactly is obfuscated. Fig. 2 and lines 4-31 in Algorithm 2 highlights how the selective extraction method works, which is composed of three main steps.

Step 1: Function encapsulation

The first step, called function encapsulation, starts by selecting individual lines of code from the behavioral description and encapsulates them into a functional operator (lines 8-11 of Algorithm 1). HLS tools have great controllability mainly through the use of pragmas. This allows the designer to control how to synthesize arrays (*e.g.*, RAM or registers), loops (*e.g.*, unrolling, partially unrolled or pipelined) and functions (*e.g.*, inline, goto or function operator). This last option is used in the proposed method. Synthesizing functions as operators allows users to encapsulate functions as functional units. The original idea behind this is to allow the user to control the amount of parallelism in the resultant circuit, as *inlining* generates a hardware block for every function call, and *goto* creates a single hardware block for the function. Encapsulating a function into an operator allows users to control the number of instantiations for the given function and thus the amount of parallelism. This is typically done by specifying the instantiation number in a constraint file.

Pragma annotation : Pragma annotation seeks to discover all *mappable* lines that qualify for obfuscation, where *mappable* implies a line that can be isolated and mapped as functional operator. We annotate the pragma to all *mappable* lines by analyzing the code syntax as 4 types:

Type 1 is a single operation line which contains an assignment instruction. This is often a major part of C code examples and often is part of the other syntax types (examples are lines 9, 12 and 15 for C_{orig} in Fig. 3).

Type 2 applies to for *loop* syntax. The start and end of a for *loop* must lie within one module, however, pragmas internal to a for *loop* can be placed in a sub-module. An example is lines 2-13 for C_{orig} in Fig. 3).

Type 3 is an $if or if-else$ condition. The start and end of an if or $if-else$ condition must lie within one module, however, pragmas internal to if or $if-else$ condition can be placed in a sub-module. An example is lines 5-10 for C_{orig} in Fig. 3).

Type 4 represents all other syntax that is not qualified to be mapped to the eFPGA, for example, the main function, global or local variable declarations, and comments.

After each line is classified into one of the above 4 types, the three pragma types are annotated as shown in lines 3-5 in Algorithm 1.

ALGORITHM 2: Selective behavioral description extraction for obfuscation in eFPGA

Define $Dobfus$ as a partitioned design (sub-module and top module) characterized by A, L, S, f_{target}
A: Design area L: Latency S: Security metric
f_{target}: HLS target frequency
input: $\{C_{orig}, Lib_{ASIC}, Lib_{eFPGA}, f_{target}\}$
C_{orig}: Behavioral description to be obfuscated
$ASIC_db$: ASIC technology library (db)
$eFPGA_db$: eFPGA technology library (db)

```
1  /* Pre-Characterization : Initialization*/
2  HLSTechLib_ASIC = gentechlib(ASIC_db);
3  HLSTechLib_eFPGA = gentechlib(eFPGA_db);
4  /* Selective Extraction: Step1 Function encapsulation */
5  Call Algorithm 1 ( Function encapsulation )
6  /*produces a set {C_obfus_single} */
7  /* Selective Extraction: Step2 Individual Extraction*/
8  for each C_obfus_single do
9      D_eFPGA =
         hls_eFPGA(C_eFPGA, f_target, HLSTechLib_eFPGA);
10     D_ASIC =
         hls_asic(C_ASIC, f_target, HLSTechLib_ASIC);
11     D_obfus_single = {D_eFPGA, D_ASIC} ;
12     if A_total > A_max) then
13         discard D_obfus_single;
14     end
15 end
16 /* Selective Extraction: Step3 Results Merging*/
17 for each C_obfus_single do
18     C_obfus_merged = C_obfus_single;
19     for each subsequent C_obfus_single do
20         /* Merging*/
21         C_obfus_merged = C_obfus_single ∪ C_obfus_merged;
22         D_eFPGA_merged =
             hls_eFPGA(C_eFPGA_merged, f_target,
23           HLSTechLib_eFPGA);
24         D_ASIC_merged =
             hls_asic(C_ASIC_merged, f_target, HLSTechLib_ASIC);
25         if A_total <= A_max) then
26             D_obfus_merged =
                 {D_eFPGA_merged, D_ASIC_merged} ;
27         else
28             C_obfus_merged − = C_obfus_single;
29         end
30     end
31 end
   output: Dobfus_opt =
           pareto − optimal{D_obfus_merged};
```

These pragma annotated lines can be isolated and mapped as a functional operator.

Function encapsulation: We group the *mappable* lines with pragmas into a new function with the appropriate attributes and types (*e.g.*, lines 12-21 of C_{obfus} in Fig. 3). Then, each *mappable* line is encapsulated using the functional operator option pragma (*e.g.*, line 12 in C_{obfus} in Fig. 3). Also, local variables inside the

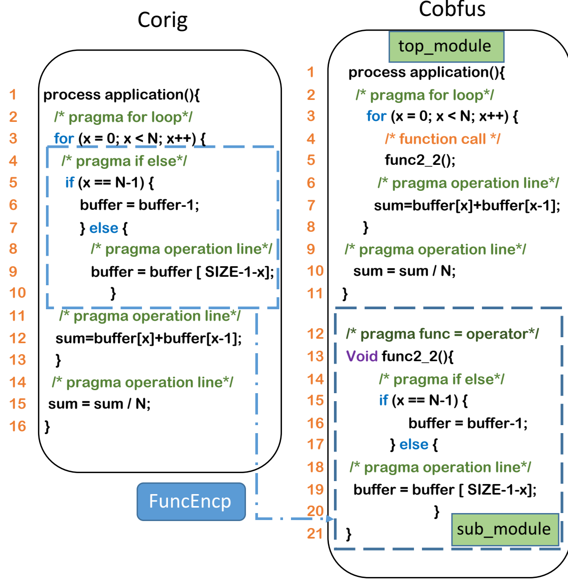

Figure 3: Example of function encapsulation.

new C function need to be declared as global variations in order to maintain the functionality. We call this step function encapsulation (FuncEncp).

Step 2: Individual Obfuscation

Since the functional operator option is used to encapsulate the portion of the behavioral description to be mapped to the eFPGA, as shown in Fig. 2, once the selected portion is automatically encapsulated into a function and the functional operator pragma specified, two separate files are generated when the behavioral description is parsed. One file with the main functionality of the application (application.c or *top_module*) and another with the part to be obfuscated (obfus.c or *sub_module*). These files can be in turn synthesized separately. The HLS proceeds with *top_module* (application.c) using the ASIC technology library ($HLSTechLib_{ASIC}$) and the encapsulated portion (obfus.c) using the eFPGA technology library ($HLSTechLib_{eFPGA}$), both generated in the library pre-characterization stage.

The result is two RTL descriptions; one for the top module (application.v) and one for the encapsulated module (obfus.v), and a report file with the quality of results (QoR_{HLS}) indicating the area of each module and latency. Our method also computes the security cost function S_i (Eq. 1) of this configuration. This allows our method to determine the overhead of mapping specific portions of code to the eFPGA quickly as well as measuring the added security.

Experimental results have shown that the QoR_{HLS} is good enough to guide our explorer in finding the best mappings, but that the actual area and delay information reported is often not too accurate (on average, we have observed differences of 20-30%). Thus, as shown in Fig. 2, the flow is extended to allow a full logic synthesis for the RTL code generated after HLS. The logic synthesis output is a gate netlist for the two modules and a new more accurate report (QoR_{LS}). This, allows us to more accurately characterize the obfuscated architecture.

Our proposed method iteratively considers each line of C code as the obfuscated portion and re-synthesizes it. If the area overhead

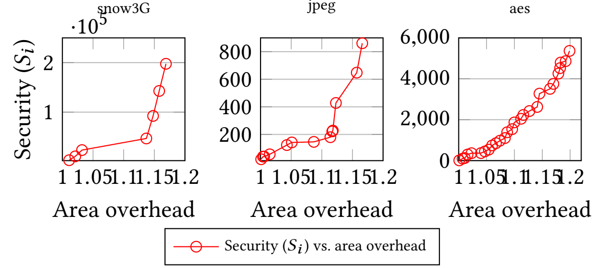

Figure 4: Area overhead vs. security cost function (Eq. 1) for pareto-optimal designs.

Table 1: Area overhead vs. security metric (TTB) for both brute-force and SAT-based attack [hours]

Benchmark	area overhead [%]	$TTB_{bfa}[hrs]$	$TTB_{sba}[hrs]$
interp	4	2^{65}	2^{56}
decim	7.8	2^{32}	2^{23}
jpeg	5.1	2^{1594}	2^{134}
kasumi	4.8	2^{1187}	2^{169}
snow3G	3.2	2^{201}	2^{119}
aes	4.7	2^{321}	2^{70}

exceeds a specified value (lines 13-15 and lines 25-29 in Algorithm 2), then this line will not be considered for inclusion in the obfuscated portion.

Step 3: Merging of Results

Our proposed method continues by merging the results obtained in step 1, whereby each result is due to the obfuscation of a single line of C code. In this step, multiple lines of C code are considered for possible concurrent obfuscation. To achieve this, the proposed method grabs each valid single obfuscated line, in turn, and iteratively seeks to merge additional subsequent obfuscated lines of C code. Once the area constraint would be violated, merging ceases and the merging process starts over with the next valid single line of obfuscated C code (lines 19-30 in Algorithm 2). Note that each single valid line of obfuscated C code is trial merged with subsequent lines of obfuscated C code until the area constraints are violated. This means that there may be considerable overlap, in terms of lines of obfuscated C code, between the various mergings. However, this design space will be significantly pruned, retaining only the Pareto-optimal points, after considering the area overhead, delay overhead and security of each merging.

6 EXPERIMENTAL RESULTS

Different computationally intensive applications, amiable to hardware acceleration, were selected in order to test our proposed method. For this purpose, six SystemC benchmarks from the freely available Synthesizable SystemC Benchmark suite S2CBench [12] were used. In particular, they are : 3-stage *interpolation* filter, 5-stage *decimation* filter, *jpeg* encoder, *snow3G* stream cipher, *kasumi* block cipher used in mobile communications and *aes*.

The HLS tool used is CyberWorkBench from NEC [6] and the target technology is Globalfoundries 65nm. The HLS target frequency is fixed in all cases to 200MHz and the logic synthesis tool used is Synopsys's Design Compiler. The experiments are conducted

Table 2: FSM increase for configurations shown in Table 1

Benchmark	obfuscated FSMs	original FSMs	increased FSMs
interp	5	4	1
decim	9	8	1
jpeg	48	46	2
kasumi	4	3	1
snow3G	4	3	1
aes	10	9	1

on an Intel i7-6700 3.50GHZ CPU and 16 GB memory, running CentOS 7.0. It should be noted that in order to get accurate results, the results presented are those reported after logic synthesis.

Fig. 4 shows the Pareto-optimal trade-off curve design space exploration results for benchmarks *snow3G*, *jpeg* and *aes* as an example of area overhead versus security cost function. The y-axis represents the security cost function (Eq. 1) when different portions of the behavioral description are mapped onto the eFPGA. The area overhead was restricted to 20 percent more than the original circuit size. It shows, as expected, a relationship between the size of the circuit mapped to the eFPGA block and the security. The larger the portions of the circuit mapped to the eFPGA block, the more secure the circuit becomes.

Table 1 lists area overhead versus the TTB security metric for both brute-force and SAT-based attacks for the six benchmarks used. We select the obfuscated designs (D_{obfus}) with the area overhead fixed at around 5% from the Pareto-optimal configurations as an example to show the TTB in hours. TTB is obtained from Eq. 2 and Eq. 3 in Section 4 where bitstream size ($bitstream_i$), logic cell number ($cell_i$) and $latency_i$ are obtained from QoR of the logic synthesis step. In each case, for all practical purposes, it would not be possible for an attacker to reverse engineer the design.

One additional dimension that Fig. 4 does not cover is the performance overhead introduced by mapping a portion of the design to the eFPGA. In order to simplify the analysis, Table 2 summarizes the increase in FSMs for configurations shown in Table 1. In all cases, the target operating frequency of 200MHz could be met, as we provided the HLS tools with the detailed technology libraries of the ASIC and eFPGA, and hence the scheduling phase of the HLS process can insert more or less logic circuits in a single step so that the 5ns maximum delay (1/200MHz) is not violated. The performance degradation is thus observed as the number of FSMs required to produce a new output, as the extra delay introduced from the eFPGA block can force more scheduling steps. Table 2 shows the FSM increase of the six benchmarks. Only one additional state is needed after obfuscation for 5 of the 6 benchmarks.

In Table 3, the second column is the number of lines of C code in the benchmarks. The third column is the number of pragmas, or candidate code lines that could be obfuscated. The fourth column is the number of lines of code implemented in the eFPGA, for the results in Table 1. The last column shows the running time of our proposed design space exploration method for the results in Table 1, where in this case the run time of the HLS process accounts for 25% of the total runtime while the logic synthesis accounts for the remaining 75%. In all cases, the TTB is so long that for all practical

Table 3: Selective extraction algorithm runtime [seconds] for the results shown in Table 1

Benchmark	# of code lines	pragma#	Obf.# of code lines	runtime
interp	155	28	22	18
decim	433	34	13	22
jpeg	482	72	62	51
kasumi	314	28	4	25
snow3G	458	73	6	26
aes	892	59	10	64

purposes the designs are secure and the area overhead is around 5%. In most cases the latency was increased by only one FSM, while the computation time are modest.

7 CONCLUSIONS

In this work we have addressed the hardware security problem using an algorithm and architecture to map selective portions of a behavioral description for HLS to an eFPGA. In this manner, only the end-user has access to the full functionality of the chip. Using six benchmark circuits, we showed that the TTB is so long (at least 8 million hours) that for all practical purposes the designs are secure, while incurring area overheads of around 5%. Further, latencies were only slightly increased, while the computation times are under one minute.

REFERENCES

[1] Achronix. 2018. Speedcore eFPGA. (2018).
[2] Yousra Alkabani et al. 2007. Active Hardware Metering for Intellectual Property Protection and Security. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium (SS'07)*. Article 20, 16 pages.
[3] R. S. Chakraborty and S. Bhunia. 2009. HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection. *IEEE TCAD* 28, 10 (Oct 2009), 1493–1502.
[4] Y. Lao et al. 2015. Obfuscating DSP Circuits via High-Level Transformations. *IEEE TVLSI* 23, 5 (May 2015), 819–830.
[5] Bao Liu et al. 2014. Embedded Reconfigurable Logic for ASIC Design Obfuscation Against Supply Chain Attacks. In *DATE*. 243:1–243:6.
[6] NEC. 2015. CyberWorkBench v.5.2. (2015).
[7] Quicklogic. 2018. ArticPro. (2018).
[8] J. Rajendran et al. 2013. Is split manufacturing secure?. In *2013 DATE*. 1259–1264.
[9] J. Rajendran et al. 2013. Security Analysis of Integrated Circuit Camouflaging. In *SIGSAC Conference on Computer & Communications Security (CCS '13)*. 709–720.
[10] J. Rajendran et al. 2015. Fault Analysis-Based Logic Encryption. *IEEE Trans. Comput.* 64, 2 (Feb 2015), 410–424.
[11] Jarrod A. Roy et al. 2008. EPIC: Ending Piracy of Integrated Circuits. In *DATE*. 1069–1074.
[12] Carrion Benjamin Schafer et al. 2014. S2CBench:Synthesizable SystemC Benchmark Suite. *IEEE Embedded Systems Letters* 6, 3 (2014), 53–56.
[13] Mustafa M. Shihab et al. 2019. Design Obfuscation through Selective Post-Fabrication Transistor-Level Programming. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE.
[14] S.Liu et al. 2013. Achieving Energy Efficiency Through Runtime Partial Reconfiguration on Reconfigurable Systems. *ACM Trans. Embed. Comput. Syst.* 12, 3 (2013), 72:1–72:21.
[15] Jingxiang Tian et al. 2017. A field programmable transistor array featuring single-cycle partial/full dynamic reconfiguration. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 1336–1341.
[16] T. Winograd et al. 2016. Hybrid STT-CMOS designs for reverse-engineering prevention. In *DAC*. 1–6.
[17] M. Yasin et al. 2016. On Improving the Security of Logic Locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 9 (Sept 2016), 1411–1424.