DECOY: DEflection-Driven HLS-Based Computation Partitioning for Obfuscating Intellectual PropertY

Jianqi Chen[1], Monir Zaman[1], Yiorgos Makris[1], R. D. (Shawn) Blanton[2], Subhasish Mitra[3] and Benjamin Carrion Schafer[1]
[1]Department of Electrical and Computer Engineering, The University of Texas at Dallas, Richardson, TX, USA
[2]Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA
[3]Department of Electrical Engineering and Department of Computer Science, Stanford University, Stanford, CA, USA
{jianqi.chen, monir.zaman, yiorgos.makris, schaferb}@utdallas.edu, rblanton@andrew.cmu.edu, subh@stanford.edu

***Abstract*—Among various competing designs targeting similar functionality, the key differentiator typically consists of a small amount of custom Intellectual Property (IP). To protect this IP from reverse engineering, designers need effective solutions for *hiding* the unique aspects of their implementations. In this work, we introduce a general framework for partitioning the computation performed by a design into a part whose implementation is commonly known (and encountered across many designs), and a part which is unique to this design. The former can then be built using conventional techniques (including untrusted manufacturing facilities) while the latter needs to be protected using additional obfuscation techniques. The existence of several other known implementations of the (same or similar) target function serves as a *decoy* which deflects efforts seeking to reverse-engineer the unique implementation. We demonstrate our framework using a hardware accelerator case study where (a) partitioning is performed through High Level Synthesis (HLS), (b) the commonly known portion of the accelerator is implemented as an Application Specific Integrated Circuit (ASIC), and (c) the unique portion of the accelerator is implemented on an embedded Field-Programmable Gate Array (eFPGA).**

I. Introduction

Today's globalized fabless model leads to serious security and trustworthiness concerns [1], as companies have to reveal their Intellectual Property (IP) to potentially untrusted entities: Electronic Design Automation (EDA) software, photolithography mask manufacturers, silicon foundries, test houses and distributors. Untrusted entities may illegally use and/or reproduce such IP for economic or other benefits. This happens at an exact moment when System-on-Chip (SoC) designs are relying on IP more than ever, for example inside hardware accelerators for better energy and speed, higher accuracy of results or unique algorithmic capabilities. Accordingly, in this paper we address the following problem: *given a commonly known function and its specific design implementation, which contains a critical IP for product differentiation and success, how do we protect this critical IP from being exfiltrated by unauthorized parties in various stages of this global ecosystem?*

Our solution relies on the following observation: **Competing designs targeting similar functionality generally use similar blocks; only a small portion serves as the differentiating IP that gives a design its competitive edge.** For example, while hardware accelerators targeting a particular domain contain similar types of computation engines and memory controllers (or even interconnect architectures), what sets them apart is the small amount of custom logic that enables enhanced functionality or combines these building blocks in unique or superior ways.

Based on this observation, we present a general framework called *DECOY*, which exploits functional similarities across various competing designs to provide mechanisms for hiding the differentiating IP from untrusted portions of the supply chain. Depending on the threat model and the location and capabilities of the adversary, existing obfuscation solutions (*e.g.,* post-manufacture programming, multi-chiplets, logic locking, layout camouflaging, split manufacturing, split design-flow) can be used for hiding. The novelty of DECOY is in its way of selecting the portion of the design that must be protected (obfuscated). DECOY considers multiple known implementations of similar functionality and identifies their commonality with the target design. The unique implementation aspects of the target design are then selected for protection (obfuscation). The key advantage of this approach is that access to an operational device (*i.e.,* oracle) does not substantially help the attacker: indeed, existence of multiple (known) options for reinstating the missing functionality deflects exact IP exfiltration (which might now face an exhaustive search).

The key contributions of this paper include:

- A general framework for obfuscating the IP that gives a design its competitive edge by deflecting reverse-engineering attacks through similar commonly-known (possibly inferior) implementations, serving as decoy.
- Demonstration of this framework for a hardware accelerator case study, where identification and partitioning of the differentiating IP is performed through High Level Synthesis (HLS), the commonly known portion is implemented as Application-Specific Integrated Circuit (ASIC), and critical IP is obfuscated through an embedded Field-Programmable Gate Array (eFPGA).

II. Motivational Example

Fig. 1 shows a motivational example using an auto-encoder artificial neural network (ANN) with a single hidden-layer. Fig. 1(a) is a typical implementation (*i*.e., C_{known}) using the sigmoid activation function, whose ASIC implementation is shown in Fig. 1(b). Fig. 1(c) shows a superior implementation (*i*.e., C_{new}) using a rectified linear unit (ReLU) activation function. ReLU is much simpler to implement in hardware (*i*.e., 99% smaller) and, for these types of ANNs, produces similar accuracy. To hide RELU from the competition, DE-

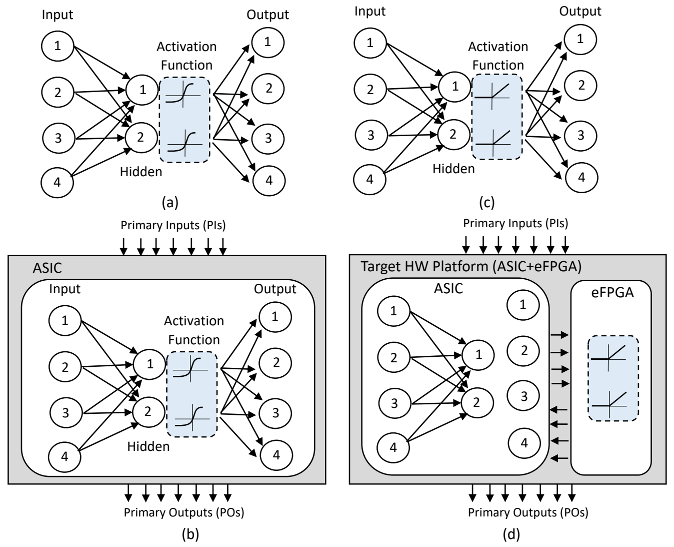


Fig. 1. Motivational example (a) Well-known implementation of Auto-encoder ANN using sigmoid activation function, (b) full implementation on ASIC, (c) New implementation of same ANN using ReLU activation function, and (d) implementation on our proposed ASCI+eFPGA platform hiding the differences between implementations.

COY maps the unique portion of the design onto an eFPGA, as shown in Fig. 1(d).

While this simple example uses only one well-known C_{known} implementation and assumes that C_{known} and C_{new} perform the same functionality, DECOY generalizes to the case of multiple C_{known} versions and similar (rather than same) C_{new} functionality. Thereby, DECOY can prioritize key IP protection objectives, such as hiding unique functionality, while at the same time enable exploration of the various trade-offs between performance, power, area and security. With this generalization in mind, we formulate our approach as follows:

Input: (1) A well-known behavioral description C_{known} for HLS which leads to a well-known implementation RTL_{known}, and (2) A new behavioral description C_{new} for HLS which leads to a proprietary/superior RTL_{new}.

Objective: Partition C_{new} into two parts (leading through HLS to RTL_{common} and RTL_{unique}) such that: (a) $RTL_{common} \cup RTL_{unique}$ can implement RTL_{new}, and (b) RTL_{common} has the same functionality as $RTL_{new} \cap RTL_{known}$.

Here, RTL_{common} and RTL_{unique} are mapped on an ASIC and an eFPGA as RTL_{ASIC} and RTL_{eFPGA}, respectively.

III. Related Work

Various techniques have been proposed to protect VLSI designs from the unlawful use and reverse engineering. These techniques can be broadly classified into either authentication-based or obfuscation-based.

Authentication-based techniques include the use of physical unclonable functions (PUFs) [2] for authentication, water-marking [3] and functional locking [4]–[6].

On the other hand, 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

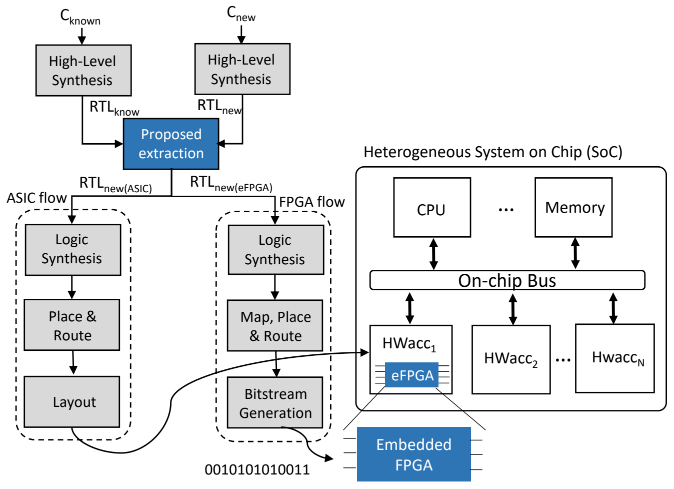


Fig. 2. Proposed obfuscation flow and target architecture.

DSP circuits by performing high-level transformations during the design stage [7].

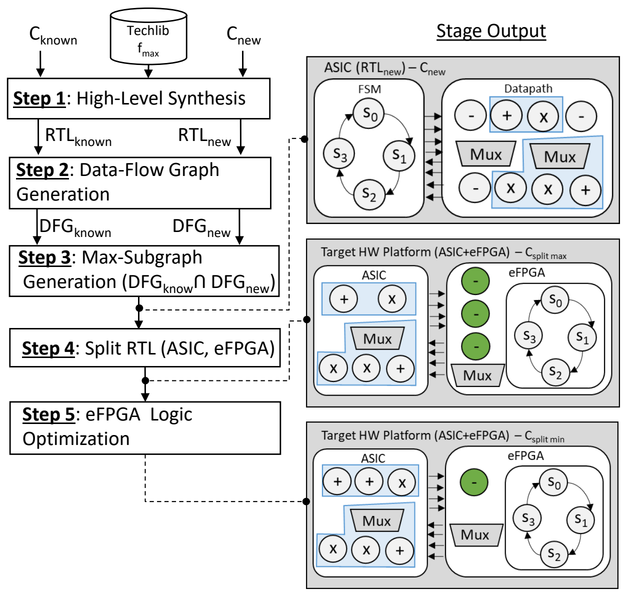
Mapping selective portions of an applications onto FPGAs has until now been mainly studied in the context of hardware acceleration [8] and [9] or energy reduction [10]

In terms of using FPGAs, or FPGA-based inspired structures for obfuscation, the authors in [11] presented a method to extract different paths at the gate-level netlist and map these gates to non-volatile spin transfer torque (STT) based reconfigurable LUTs. Although a promising approach, commercial production of these hybrid circuits is not yet feasible. In [12], the authors proposed a mechanism for structurally obfuscating sensitive parts of a design given in Verilog through a custom, dedicated fine-grained reoconfigurable fabric. The benefits of having their own custom ultra-fine grained fabric vs. other types of reconfigurable fabric are, nevertheless, not clear, considering the overhead in the bits required to reconfigure such a FPGA. Very recently, the authors in [13], studied the effect of LUT sizes on the obfuscation strength. Closer to our work, in [9], the authors use an embedded FPGA to cost-effectively obfuscate *predetermined* portions of a design, given as a behavioral description for HLS.

To the best of our knowledge, this is the first work that addresses the issue of obfuscation, driven by the existence of known other implementations [14].

IV. Target Hardware Platform

As shown in Fig. 1(d), the target hardware platform envisioned in this work is composed of: (i) an ASIC portion onto which the common logic between the well-known implementation and the new implementation of an application is mapped, and (ii) an eFPGA portion to map the distinct logic, which is unique to the new implementation. The latter portion of the design can be fully obfuscated in the eFPGA because the silicon foundry does not have access to the bitstream that configures the eFPGA during the fabrication process. Several fine-grained eFPGA providers allow full customization of the reconfigurable fabric [15], [16]. Fig. 2 shows the complete overview of the proposed VLSI design flow and the target



C_{known}
Techlib
f_{max}
C_{new}
Stage Output
Step 1: High-Level Synthesis
RTL_{known}
RTL_{new}
Step 2: Data-Flow Graph Generation
DFG_{known}
DFG_{new}
Step 3: Max-Subgraph Generation ($DFG_{know} \cap DFG_{new}$)
Step 4: Split RTL (ASIC, eFPGA)
Step 5: eFPGA Logic Optimization
ASIC (RTL_{new}) – C_{new}
FSM
Datapath
Mux
Target HW Platform (ASIC+eFPGA) – $C_{split\ max}$
ASIC
eFPGA
Target HW Platform (ASIC+eFPGA) – $C_{split\ min}$

Fig. 3. Overview of complete proposed flow (DECOY).

architecture, mainly targeting hardware accelerators in the SoC.

The target architecture is a hardware accelerator mapped as an ASIC onto a heterogeneous SoC, although it could also be a standalone block as well. The overall flow starts by comparing two implementations of the same application, given as untimed behavioral descriptions for HLS. The main contribution of this work is to partition the new implementation that needs to be hidden into the target hardware platform, such that the newly developed part of the design is mapped onto the eFPGA and the well-known part onto the ASIC. The RTL for the eFPGA is then passed to the FPGA synthesis flow, which ends up generating a bitstream file for the eFPGA, while the RTL for the ASIC follows a traditional ASIC flow, as shown in Fig. 2.

V. Proposed Method

Fig. 3 shows an overview of the proposed method, which we call **DECOY**, while algorithm 1 describes the core steps of the proposed method in detail (Steps 2 to 4). Our method takes two behavioral description (C_{known} and C_{new}) as input, that implement the same functionality in different ways. C_{known} is the traditional implementation, while C_{new} is the new implementation that a company wants to hide from the competition. Our proposed method then synthesizes (HLS) each behavioral description and obtains the two RTL descriptions that can efficiently execute them (RTL_{known} and RTL_{new}). HLS typically generates a circuit that is composed of a finite state machine (FSM) and a datapath. The FSM serves as the dataflow controller. The datapath consists of functional units (FUs, *e.g.,* adders, multipliers and dividers), muxes, decoders and memory from the user-specified HLS libraries.

The proposed method, then, continues by building a Data Flow Graph (DFG) of the datapath for the two design implementations (DP_{known} and DP_{new}). Next, it compares between the dataflow graphs and extracts the common functional units (FUs). These common FUs serve as the starting points for extracting the common logic between the two implementations. The proposed method, then, continues by grouping RTL components from these common FUs. The main idea is to generate the largest common subgraphs between the two DFGs, starting from the common FUs. These common subgraphs are, in turn, mapped onto the ASIC, while the rest of the DP_{new} is mapped onto the eFPGA. In summary, the proposed method can be divided into 5 steps as follows:

Step 1: High-Level Synthesis: This first step takes the application as input in its well-known form (C_{known}) and its new implementation that needs to be hidden (C_{new}). The inputs are synthesized to equivalent RTL codes and C_{new} is mapped as the hardware accelerator onto an SoC. In order to accelarate the execution of this application, the HLS is set to maximize performance by unrolling loops, and inlining functions.

Step 2: Data Flow Graph Extraction: This step generates the DFGs for the two RTL descriptions obtained in step 1 (DFG_{known} and DFG_{new}) by parsing the generated RTL codes from PIs to POs (algorithm 1: lines 1 to 3). In this case, DFG_{known} and DFG_{new} are modelled as a graph $G(V, E)$, where a node $v \in V$ represents a RTL component $V = \{FU, mux, decoder, encoder, RAM\}$, where $FU = \{+, *, >, <, ...\}$. The connection between these components is represented by an edge $e(v_i, v_j) \in E$ with v_i driving v_j. Because the typical circuit after HLS is composed of $FSM + Datapath$, our method also includes the FSM in the DFG.

Step 3: Max-Subgraph Generation: This third step parses both DFGs and constructs the largest common subgraph between the two DFGs. The construction method starts by identifying in the two dataflows (DFG_{known} and DFG_{new}) a common set of FUs (algorithm 1: line 5), such that $FU_{common} = \{rtl_1, rtl_2, \ldots, rtl_N\} \in DFG_{known} \cap DFG_{new}$. The method then randomly chooses one of the common RTL components (rtl_i) and builds the largest common subgraph (algorithm 1: lines 10 to 24). Although max-subgraph generation has been shown to be an NP-complete problem [17], the RTL code does not generally contain a larger number of components, as opposed to building max-subgraphs at the gate netlist-level. Thus, our method uses a depth-first method to traverse the two DFGs, in order to build the max-subgraph. The method continues with the next common RTL component (rtl_{i+1}) and repeats the same process, unless this common RTL component has already been included in one of the subgraphs (algorithm 1: lines 7 to 9).

Step 4: RTL Partition: This step splits the RTL description of the new design (RTL_{new}) into the ASIC and eFPGA part, based on the max-subgraph generated in step 3, where the the common subgraph is mapped onto the ASIC portion of the design, while the rest of the circuit is mapped to an eFPGA (algorithm 1: line 26). The method then proceeds to synthesize these descriptions separately, given the ASIC

Algorithm 1: ASIC+eFPGA partitioning approach

```
input : $RTL_{known}, RTL_{new}$
    $RTL_{known}$: Known RTL implementation of application
    $RTL_{new}$: Novel RTL implementation of same application
output: $RTL_{partitioned} = RTL_{ASIC} \cup RTL_{eFPGA}$
    $RTL_{partitioned}$: Split design between ASIC and FPGA
/* Step 2: DFG Generation*/
$DFG_{known} = parse\_rtl(RTL_{known})$
$DFG_{new} = parse\_rtl(RTL_{new})$
/* Step 3: Max-subgraph generation*/
$FU_{common} = search\_FUs(DFG_{known}, DFG_{new})$
/* Generate Common RTL components subgraph */
for ($rtl_i \in FU_{common}$) do
    DFS($rtl_i$)
end
Function DFS ($rtl_i$):
    if $rtl_i \notin RTL_{common}$ then
        $RTL_{common} \leftarrow RTL_{common} \bigcup \{rtl_i\}$
    else
        return
    end
    for $rtl_j$ connected to $rtl_i$ in $DFG_{new}$ do
        for $rtl_k$ connected to $rtl_i$ in $DFG_{known}$ do
            if $rtl_j = rtl_k$ and $rtl_k$ is not visited then
                mark $rtl_k$ as $visited$
                $DFS(rtl_j)$
            end
        end
    end
    return
/* Step 4: RTL Partition*/
$RTL_{unique} = extract\_RTL(DFG_{new}, RTL_{common})$
$RTL_{partitioned} \leftarrow RTL_{unique} \cup RTL_{common}$
return $RTL_{partitioned}$
```

and eFPGA libraries, to obtain the final gate-level design and eFPGA bitstream. The overhead associated with our proposed method is fully made observable at this point. In Fig. 3, it can be observed that the adders, multipliers and one mux are shared between the two implementations and are, hence, mapped onto the ASIC, while the FSM, the subtractors and the other mux are not, and are, hence, mapped to the eFPGA.

Step 5: eFPGA Logic Optimization: This last step aims at reducing the logic mapped onto the eFPGA, as this is the main source of area, delay and power overhead. For this, a proposed optimization pass makes use of a well-known optimization technique used in HLS: resource sharing. In resource sharing, multiple operations in the C code are mapped onto the same FU in order to reduce the area of the final circuit, albeit increasing the resultant latency. Thus, this optimization pass selects all the FUs mapped onto the eFPGA, each defined by a 2-tuple with the type and number of FUs, *e.g.,* $FU_{eFPGA} = \{(+, x), (*, y), \ldots, (/, z)\}$, where the eFPGA has x adders, y multipliers and z dividers. The optimization pass back-annotates the number of FUs allowed in the FU constraint file of the HLS tool, such that only a single FU of each type mapped onto the eFPGA is allowed, leading to: $FU_{eFPGA} = \{(+, 1), (*, 1), \ldots, (/, 1)\}$. In case FUs of the same type but different bitwidths are present, then only the FU of the largest bitwidth is kept. This minimizes the total area of the eFPGA by forcing the HLS process to maximize resource sharing. The resultant performance degradation will depend on the number of FUs saved. The C code is, in turn, re-synthesized (HLS) with the new constraint file, and steps 2 and 3 are repeated. This should lead to a new partitioning with fewer components mapped to the eFPGA and, hence, smaller area and power overhead (C_{split_min}). The designer can then decide which of the two configurations to choose from, based on his/her preferences. In Fig. 3, it can be observed that after step 4, the eFPGA contains three subtractors, thus, $FU_{eFPGA} = \{(-, 3)\}$. After this optimization pass, the number of subtractors allowed is only one, *i.e.,* $FU_{eFPGA} = \{(-, 1)\}$, which reduces the total eFPGA area usage.

VI. Security Analysis

In its $ASIC + eFPGA$ incarnation, DECOY defends against efforts to exfiltrate the proprietary IP in both of the following scenarios: (a) when the adversary only possesses an unprogrammed device (and all its design details); and (b) when, in addition, the adversary also possesses a programmed device that can be interrogated (*i.e.,* oracle) but from which the program cannot be extracted. In fact, access to an oracle does not offer an advantage in this case, as the adversary already has access to known implementations of the missing logic. Therefore, the traditional SAT-attack formulations may not be effective in this case. Indeed, the ability to identify all functionally equivalent solutions for the obfuscated IP (*i.e.,* through an ALL-SAT formulation) and select among them the correct one (*e.g.,* through parametric analysis) would be required. Such attack solutions, however, can be computationally prohibitive (and are yet to be developed and investigated by the hardware security community). Even for instances where ALL-SAT may be tractable, such as small or simple IP for which the cardinality of the solution set (*i.e.,* number of possible implementations) is small (*e.g.,* polynomial), the general DECOY framework provides novel IP protection opportunities. For example, DECOY can drive not only the selection of the protected IP functions but also (i) their transformation based on ALL-SAT hardness criteria, such as the existence of multiple known implementations of identical or similar functionality, and (ii) their partitioning to trusted and untrusted portions of the supply chain, through split design flow, split manufacturing, or multi-chip(let) implementations.

VII. Experimental Results

To test the effectiveness of the proposed method, we implemented two versions of well-known applications in C. The first design corresponds to the known implementation of the application (C_{known}), while the second design represents the new implementation (C_{new}). This version is also in all cases the more efficient version. In particular, consider a convolutional neural network (CNN) using sigmoid as activation function (C_{known}) and a CNN using ReLU (C_{new}). This CNN is designed for traffic sign recognition, which takes in 1280×720 grayscale images and outputs the location of the traffic sign and the corresponding speed limit. The only difference between the two implementations is that C_{new}

uses ReLU as activation function and C_{known} uses sigmoid. The second application is sorting algorithms, where we used insertion-sort as C_{known} for arrays of 100 elements and quick-sort as C_{new}. The third application used is 4×4 matrix inversion using the adjoint method as C_{known} and using LU decomposition as C_{new}. Moreover, a 16-point discrete Fourier transform using a naïve algorithm (C_{known}) and fast Fourier transform (C_{new}) is used. Finally, a naïve string matching algorithm (C_{known}) vs. Rabin-Karp algorithm (C_{new}) is also used in our experiments.

The HLS tool used is CyberWorkBench from NEC [18]. We use Design Compiler for ASIC and Quartus II for FPGA design synthesis. The Nangate 45nm technology is used for the ASIC and Stratix III as the target FPGA device that mimics the proposed eFPGA. We targeted the Stratix III because it is reported to be fabricated in the same technology node. The experiments are conducted on an Intel i7-6700 3.40GHZ CPU with 16 GB memory, running CentOS 7.

Table I summarizes the experimental results when both versions are synthesized targeting highest-performance as the main design objective. This is also the most natural way to implement these applications, as the final goal is to map them as hardware accelerators in an SoC. The $C_{known}ASIC$ column reports the area, number of RTL components, latency and delay of the known implementation, fully-mapped onto the ASIC. The $C_{new}ASIC$ column reports, for reference, the same information when the new version is fully mapped onto an ASIC.

Finally, in Table I, the C_{split_max} column represents various design metrics for the DECOY method when maximum number of FUs are allowed during C_{new} synthesis. Similarly, the C_{split_min} column reports the overall design metrics for the DECOY method when eFPGA logic optimization (step 5) is applied. We use μm^2 for ASIC area and ALUTs for FPGA area representation. To better illustrate the results, Fig. 4 graphically compares the normalized area, latency, delay, throughput, power and energy overheads of the four implementations, while the last entry in each figure shows the average results. For area, the ALUTs were converted into μm^2 following the indications of [19].

Several interesting observations can be drawn from these results. First, the new implementation of the well-known application (C_{new}) always leads to better performance than the traditional implementation. The average throughput increases by 2.62×. This is inline with the main motivation behind the proposed method. Moreover, although mapping portions of the new design to the eFPGA leads to performance degradation, it can be observed that the throughput of the proposed split-design is still higher than the throughput of the original implementation. On average by 1.42× for C_{split_max} and 1.19× for the C_{split_min}.

Second, the area overhead of DECOY can be relatively large, depending on the amount of overlap between the original implementation and the new one. This area overhead can be substantially reduced, anyway, through the proposed resource sharing optimization pass (step 5), as described in Section V.

TABLE I
EXPERIMENTAL RESULTS SUMMARY

Convolutional Neural Network using ReLU vs. Sigmoid						
	C_{known}	C_{new}	C_{split_max}		C_{split_min}	
	ASIC	ASIC	ASIC	FPGA	ASIC	FPGA
Area*	186,320	82,749	81,885	472	81,885	472
# rtl_{dp} **	1,726	1,405	1,404	1	1,404	1
Latency [clk cycle]	238,803,508	21,000,394				
Delay [ns]	14.76	11.13	18.1		18.1	
Quick Sort vs. Insertion Sort						
	C_{known}	C_{new}	C_{split_max}		C_{split_min}	
	ASIC	ASIC	ASIC	FPGA	ASIC	FPGA
Area*	12,909	29,755	8,972	2,895	11,314	2,088
# rtl_{dp} **	224	457	169	288	216	237
Latency [clk cycle]	5,390	1,088			1,442	
Delay [ns]	2.3	4.49	6.94		7.56	
Matrix Inversion using LU Decomposition vs. Adjoint Method						
	C_{known}	C_{new}	C_{split_max}		C_{split_min}	
	ASIC	ASIC	ASIC	FPGA	ASIC	FPGA
Area*	78,295	31,327	27,799	4,419	23,271	3,339
# rtl_{dp} **	503	217	185	32	135	17
Latency [clk cycle]	293	41			44	
Delay [ns]	14.11	9.95	35.33		40.34	
Fast Fourier Transform vs. Naive Discrete Fourier Transform						
	C_{known}	C_{new}	C_{split_max}		C_{split_min}	
	ASIC	ASIC	ASIC	FPGA	ASIC	FPGA
Area*	258,887	61,996	27,682	6,805	43,143	4,703
# rtl_{dp} **	718	449	278	171	311	134
Latency [clk cycle]	200	97			114	
Delay [ns]	13.13	10.81	12.33		13.62	
Rabin-Karp String Matching vs. Naive Algorithm						
	C_{known}	C_{new}	C_{split_max}		C_{split_min}	
	ASIC	ASIC	ASIC	FPGA	ASIC	FPGA
Area*	7,072	7,533	5,508	944	3,582	656
# rtl_{dp} **	122	144	107	37	82	32
Latency [clk cycle]	5,433	3,421			3,650	
Delay [ns]	3.26	4.9	16.89		15.56	

*unit of area: μm^2 for AISC, ALUTs for FPGA.
**number of RTL components in datapath.

On average the area increased by a factor of 7.48× and 5.79× for C_{split_max} and C_{split_min}, respectively. This area increase only affects the hardware accelerator, which is a single component in an otherwise large SoC. Moreover, the eFPGA can be easily reused for other tasks as it is fully reconfigurable at runtime.

Finally, in terms of power and energy consumption, it can be observed that the proposed method also leads to power and energy overheads due to the larger power consumption of the eFPGA compared to the ASIC. On average, the power increases by 4.55× for C_{split_max} and 3.65× for C_{split_min}, and the energy by 1.91× and 1.75× for C_{split_max} and C_{split_min}, respectively, when compared to C_{known}. One advantage of using the eFPGA is that the power density decreases. It has been reported that this leads to lower temperatures in the chip and, hence, to longer life times [20].

In summary, our method is an interesting solution for hiding a new implementation of a well-known application such that the overhead is still manageable, especially considering the added security offered.

VIII. CONCLUSION

DECOY obfuscates IP by separating a target design into a portion that overlaps with one or more commonly known implementations (targeting similar functionality) and a portion that provides a competitive edge. Its effectiveness was demonstrated through a single HLS-based incarnation of this method, wherein the former is mapped onto an ASIC while the latter is programmed on an eFPGA. To understand the

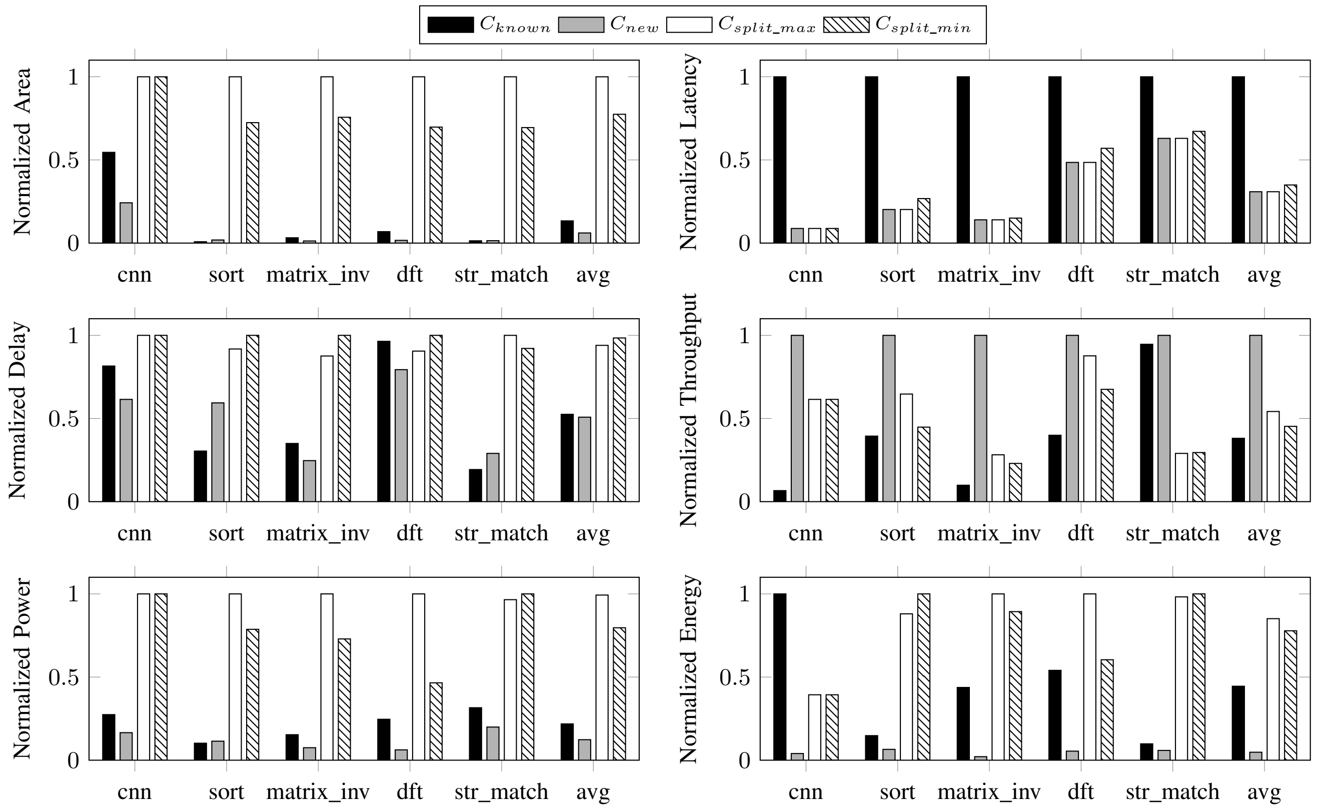


Fig. 4. Normalized area, latency, critical path delay, throughput, power, energy of C_{known}, C_{new} mapped on ASIC, and C_{split_max}, C_{split_min} mapped on ASIC and eFPGA.

full scope of DECOY, however, several research directions must be further explored. These include (i) implementation of other incarnations of DECOY wherein the above separation is achieved through split design and split design flow, split manufacturing, multi-chip(let) integration, and explorations of the various trade-offs, (ii) analysis of the impact of amount, granularity, and dispersion of such separation on IP protection (and corresponding security vs. cost tradeoffs), and (iii) investigation of new attacks which can distinguish a specific implementation among many functionally equivalent versions, as well as mitigation strategies to thwart such attacks.

References

[1] SEMI, "IP Challenges for the Semiconductor Equipment and Materials Industry," 2012. [Online]. Available: http://stg7.semi.org/sites/semi.org/files/docs/2012_IP_White_Paper_V2_SupAdd.pdf

[2] C. Herder, M. Yu, F. Koushanfar, and S. Devadas, "Physical Unclonable Functions and Applications: A Tutorial," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1126–1141, Aug 2014.

[3] Min Ni and Zhiqiang Gao, "Watermarking System for IC Design IP Protection," in *International Conference on Communications, Circuits and Systems*, 2004.

[4] J. A. Roy, F. Koushanfar, and I. L. Markov, "EPIC: Ending Piracy of Integrated Circuits," in *Proceedings of the conference on Design, automation and test in Europe*, 2008.

[5] M. Yasin *et al.*, "What to Lock?: Functional and Parametric Locking," in *GLSVLSI*, ser. GLSVLSI '17, 2017, pp. 351–356.

[6] A. Baumgarten, A. Tyagi, and J. Zambreno, "Preventing IC Piracy Using Reconfigurable Logic Barriers," *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 66–75, Jan. 2010.

[7] Y. Lao and K. K. Parhi, "Obfuscating DSP Circuits via High-Level Transformations," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 5, pp. 819–830, May 2015.

[8] N. Shirazi, W. Luk, and P. Y. K. Cheung, "Automating production of run-time reconfigurable designs," in *FCCM*, 1998, pp. 147–156.

[9] Hu *et al.*, "Functional Obfuscation of Hardware Accelerators Through Selective Partial Design Extraction Onto an Embedded FPGA," in *GLSVLSI*, 2019.

[10] S. Liu, R. N. Pittman, A. Forin, and J.-L. Gaudiot, "Achieving Energy Efficiency Through Runtime Partial Reconfiguration on Reconfigurable Systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 3, pp. 72:1–72:21, 2013.

[11] T. Winograd, H. Salmani, H. Mahmoodi, K. Gaj, and H. Homayoun, "Hybrid STT-CMOS Designs for Reverse-engineering Prevention," in *Proceedings of the Annual Design Automation Conference*, June 2016.

[12] Shihab *et al.*, "Design Obfuscation through Selective Post-Fabrication Transistor-Level Programming," in *DATE*, 2019.

[13] Kolhe *et al.*, "Security and complexity analysis of lut-based obfuscation: From blueprint to reality," in *ICCAD*, 2019, pp. 1–8.

[14] S. Dupuis and M.-L. Flottes, "Logic locking: A survey of proposed methods and evaluation metrics," *Journal of Electronic Testing*, vol. 35, no. 3, pp. 273–291, 2019.

[15] Achronix, "Speedcore eFPGA," 2019. [Online]. Available: www.achronix.com

[16] Quicklogic, "ArticPro eFPGA," 2019. [Online]. Available: www.quicklogic.com

[17] V. Kann, "On the approximability of the maximum common subgraph problem," in *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 1992, pp. 375–388.

[18] NEC, "Cyberworkbench v.6.1," 2018.

[19] H. Wong, V. Betz, and J. Rose, "Comparing FPGA vs. Custom Cmos and the Impact on Processor Microarchitecture," in *FPGA*, 2011.

[20] P. Sundararajan, A. Gayasen, N. Vijaykrishnan, and T. Tuan, "Thermal Characterization and Optimization in Platform FPGAs," in *ICCAD*, 2006.