Transient Key-based Obfuscation for HLS in an Untrusted Cloud Environment

Hannah Badier and Jean-Christophe Le Lann
ENSTA Bretagne, Lab-STICC, Brest
hannah.badier@ensta-bretagne.org, lelannje@ensta-bretagne.fr

Philippe Coussy and Guy Gogniat
Universite de Bretagne Sud, Lab-STICC, Lorient
philippe.coussy@univ-ubs.fr, guy.gogniat@univ-ubs.fr

***Abstract*—Recent advances in cloud computing have led to the advent of Business-to-Business Software as a Service (SaaS) solutions, opening new opportunities for EDA. High-Level Synthesis (HLS) in the cloud is likely to offer great opportunities to hardware design companies. However, these companies are still reluctant to make such a transition, due to the new risks of Behavioral Intellectual Property (BIP) theft that a cloud-based solution presents. In this paper, we introduce a key-based obfuscation approach to protect BIPs during cloud-based HLS. The source-to-source transformations we propose hide functionality and make normal behavior dependent on a series of input keys. In our process, the obfuscation is transient: once an obfuscated BIP is synthesized through HLS by a service provider in the cloud, the obfuscation code can only be removed at Register Transfer Level (RTL) by the design company that owns the correct obfuscation keys. Original functionality is thus restored and design overhead is kept at a minimum. Our method significantly increases the level of security of cloud-based HLS at low performance overhead. The average area overhead after obfuscation and subsequent de-obfuscation with tests performed on ASIC and FPGA is 0.39%, and over 95% of our tests had an area overhead under 5%.**

***Index Terms*—High-Level Synthesis, Cloud, IP theft, Obfuscation**

I. Introduction

Today's hardware designs have reached a fantastic degree of complexity. To sustain this industrial challenge, design flows are increasingly distributed, with companies relying on third parties for IP development, manufacturing and testing. The recent surge in cloud computing capabilities offers new opportunities to distribute design even further, by outsourcing complex computations to dedicated Software as a Service (SaaS) platforms. In the next years, Electronic Design Automation (EDA), which is compute-intensive, is likely to benefit directly from such services. In particular, HLS lends itself to being offered as a cloud-based service. However, broad adoption by the industry is slowed down by legitimate security concerns about IP theft [1]. Indeed, HLS in the cloud services have to be considered as untrusted platforms, possibly vulnerable to insider threats as well as outside attacks and security breaches. A company using such a service would have to trust the platform with its high-level source code. While the files could be encrypted during transfer to prevent Man-In-The-Middle (MITM) attacks, HLS using industry-standard products such as Xilinx Vivado HLS or Catapult from Mentor Graphics can only be performed on valid, plain-text code, meaning that the files have to be decrypted on the server. This leads to vulnerable points both before and after HLS, as depicted on Fig.1, where Behavioral Intellectual Property (BIP) could be stolen.

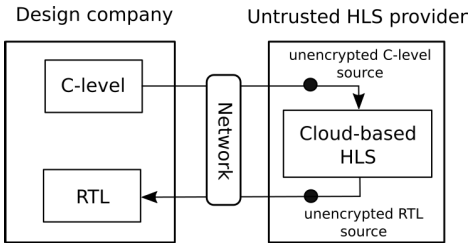

Fig. 1: HLS in the cloud design flow with vulnerable points

Our long-term goal is to enable hardware design companies to safely use a cloud-based HLS service, without having to entrust their security to the service provider. To achieve this, we aim at preventing BIP theft and reuse, before and after HLS in a cloud-based context. These security measures should be applied on the design company's side at algorithmic level, and should not disrupt the HLS process or permanently alter the functionality of the code. Our approach should also have a negligible impact on design performances.

Obfuscation is widely used as a means to protect intellectual property in software and hardware design. According to the definition in [2], an obfuscated program is functionally equivalent to the original but more difficult to understand. In this paper, we propose to combine software obfuscation techniques that make the code harder to understand with key-based hardware obfuscation techniques that prevent correct execution and thus reuse of the code. If the design company can provide the correct obfuscation keys, our process removes obfuscation code at register transfer level (RTL) and restores original functionality with low design overhead. In particular, the main contributions of this paper are the following:

- We consider a hardware design flow where HLS is used as a cloud service and where security does not rely on trust in the cloud provider.
- We propose a novel low-overhead method for BIP protection through key-based obfuscation.
- We offer a fully automated toolset, KaOTHIC, for obfuscation and de-obfuscation.

978-3-9819263-2-3/DATE19/©2019 EDAA

This paper is organized as follows. Section II discusses works related to IP protection and presents background on obfuscation. In Section III, the proposed approach is presented in detail. Section IV contains the experimental setup and discusses the results. Conclusions and information about future work are provided in Section V.

II. Related Work and Background

A. IP Protection

Due to the growing issue of IP infringement, several IP protection techniques have been studied, at different design steps and different abstraction levels. Most of them aim at protecting IP during manufacturing, which is nowadays usually outsourced to third-party foundries. Watermarking and split manufacturing (e.g. [3] [4] [5]) have been widely studied. Many solutions for obfuscation-based approaches have also been proposed. Some focus on low-level techniques based on hardware properties and applied at layout level to prevent netlist extraction, for example by using Physical Unclonable Functions (PUFs) or camouflage gates. Several techniques for IP protection through obfuscation at gate level by adding extra gates to the circuit have also been proposed. However, these techniques do not protect so-called soft IPs at register transfer level (RTL). Several methods have been proposed to add key-based obfuscated to VHDL or Verilog code, either directly at RTL or during HLS. In [6], a mode-control finite state machine (FSM) is added to the design, so that the circuit only functions in normal mode if the correct key sequence is known. In [7] and [8], high-level transformations are used to protect digital signal processing (DSP) circuits. In [9] and [10], key-based obfuscation is added during HLS by extending the HLS process. In these works, the goal is to obtain obfuscated RTL and to provide protection during manufacturing or for third-party vendors of IPs, and HLS is considered to be a trusted design step. However, they do not protect behavioral IPs before and during HLS.

In [11], a heuristic approach for optimal obfuscation of behavioral IPs is presented. It is based on a study of the impact of commercial or free software obfuscators on HLS quality of results. This method focuses on how to optimally apply software obfuscation techniques to code before HLS and does not modify functionality of the code. While it can increase the difficulty of an attacker to understand stolen BIP, it does not prevent "black-box" usage, where the BIP is directly reused or critical information about it is found by simply analyzing its inputs and outputs. In our approach, we aim at preventing both IP theft and reuse. To achieve this, we rely on key-based obfuscation techniques. However, the methods presented in [11] are compatible with our approach, which means that both can be jointly used to increase the level of protection.

To the best of our knowledge, there is no work yet using key-based obfuscation to protect BIPs in a cloud-based context.

B. Software Obfuscation

The taxonomy given in [2] distinguishes between three different types of software obfuscation. "Layout" obfuscation affects human readability of the code, while "data" and "control flow" obfuscation respectively modify data (for example encoding) and control flow (for example by splitting functions). These techniques are commonly gathered under the name of code-oriented obfuscation, as opposed to model-oriented or cryptographic obfuscation, which aims at protecting the code in a formally verifiable way through cryptographic operations. While cryptographic obfuscation offers a provably secure way of protecting intellectual property, it is not usable in a day-to-day context yet [12] [13]. In this work, we focus on code-oriented obfuscation techniques.

III. Proposed Approach

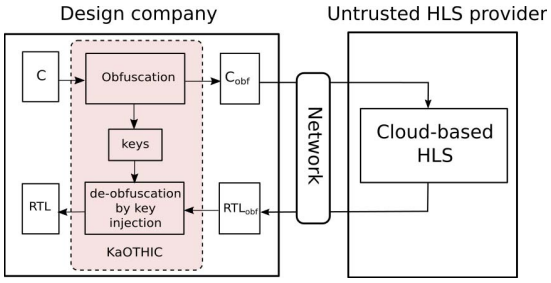

Fig. 2: HLS in the cloud design flow secured by obfuscation

A. Secure HLS in the Cloud Design Flow

We consider a hardware design company desiring to synthesize a design based on an innovative algorithm by using a cloud-based HLS service. Complete security of this service cannot be guaranteed and the design company faces the risk that a malicious adversary gains access to its C level code. This adversary could be either an insider of the HLS service provider or an outside attacker. Following risks must be considered in case of an attacker stealing the code:

1) Espionage: the attacker gains information about what type of algorithms or applications the design company is working on.
2) BIP theft: the attacker is able to understand the algorithm and can counterfeit or modify it.
3) Black-box usage: the attacker does not completely understand the algorithm, but can execute the code and reuse it.

Because we cannot prevent code theft on the HLS provider's side, our approach aims at minimizing the previously enumerated risks. We secure a traditional HLS design flow by adding two additional steps on the design company's side, before and after HLS, as illustrated on Fig.2. Protection is thus transient. The tool implementing the approach we propose is open-source, which means that security cannot rely on secrecy of the process. To prevent an attacker from reproducing obfuscation results and thus gaining valuable information on how to de-obfuscate, we aim at introducing elements of randomness wherever possible in the obfuscation flow.

The first step, detailed in Section III-B, consists in obfuscating a given C code, thus making it harder to understand. Key-based obfuscation is used to prevent black-box usage. The obfuscated C code is then sent to the cloud provider, where HLS is performed. The resulting RTL code, which is automatically also obfuscated, thanks to the code transformations we apply, is returned to the design company. If the RTL code is directly synthesized at this point, the resulting circuit will not function correctly because the logic added by obfuscation is also synthesized. This is why de-obfuscation by key injection, detailed in Section III-D, is required as a second step. De-obfuscating with the correct keys both ensures that the original functionality is recovered and that, once the circuit is synthesized, the overhead is kept at a minimum.

B. Obfuscation Flow

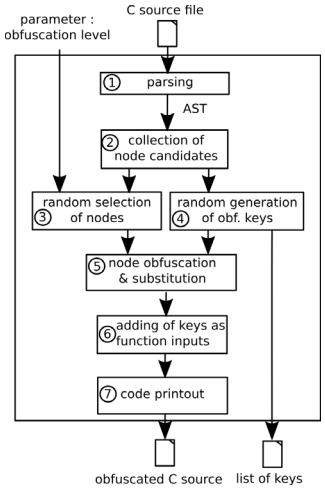

Fig. 3: Obfuscation flow

Fig.3 shows the obfuscation flow we propose. The inputs are the original source files written in C language that must be obfuscated and the obfuscation level. The outputs are the obfuscated code and a list of obfuscation keys. The process can be parameterized: the user can choose which obfuscation techniques to apply and in what order, as well as the desired obfuscation level, which is defined as the percentage of obfuscated expressions.

The flow is composed of the following steps (see Fig.3):

1) The C files are parsed and an Abstract Syntax Tree (AST) is generated;
2) A visitor pattern is used to traverse the AST and collect the list of AST nodes that are potential obfuscation candidates;
3) Based on the obfuscation level, nodes are randomly selected among the candidates;
4) One random key per selected node is generated;
5) Each selected node is obfuscated using the technique presented in the next Subsection and replaced in the original code;
6) The key variable names are added as inputs to the main function;
7) The obfuscated code is returned.

For a higher level of security, a different key is used for each obfuscated node. This ensures that an adversary guessing or discovering the value of one key does not compromise the secrecy of the other keys. To avoid replay by an attacker, the selection of candidate nodes is also done randomly.

The implemented transformations at this point include control flow flattening, control flow splitting and encoding of literals. Several other techniques can be added at a later point to increase obfuscation resistance and security. A detailed presentation of how control flow splitting is realized can be found in the next Section.

C. Control Flow Splitting by Key-based Predicate Insertion

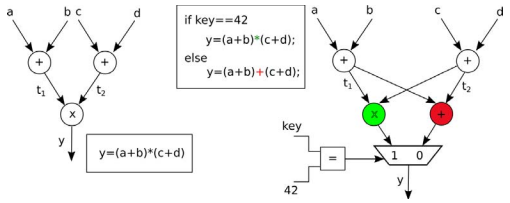

Fig. 4: Simplified example of a combinational DFG before and after obfuscation. In this example, the correct code only gets executed if ”42” is given as input key.

This technique is based on ideas presented in [2]. The control flow is split at certain points in the code by adding a predicate. A branch with bogus code is then inserted. This bogus code strongly resembles the original code, in order to increase obfuscation stealth by making it harder for an attacker to distinguish original from bogus code, but presents enough differences to the real code to modify the circuit's behavior. Usually opaque predicates, introduced in [14], whose values are known at obfuscation time but difficult to evaluate for an attacker, are used. In our case, we use predicates depending on keys that are inputs to the circuit. An example of this obfuscation technique is presented in Fig.4.

1) Predicate: For now, the predicate tests an input value against a constant. Depending on how this test evaluates, the real or the bogus code is executed. In this simplified example (Fig.4), the input is compared to the value ”42”. If the test is True, the correct code branch is executed. If any other value is given as input, the bogus code is executed. During obfuscation, we randomly assign either the original code branch or the bogus one to execute when the predicate is true. For an attacker, there are thus 2 possibilities to test. Either the input key is equal to the constant, or it is not. For N obfuscated expressions, there are thus N keys and 2^N possibilities to test in a brute-force attack to recover the original code. We assume that a high number of expressions will be obfuscated using this technique in each source code, resulting in a high number of possibilities to test for an attacker. Difficulty of a brute-force attack can be easily increased by adding more than just one bogus branch per obfuscated expression, with the key input being tested against several constant values.

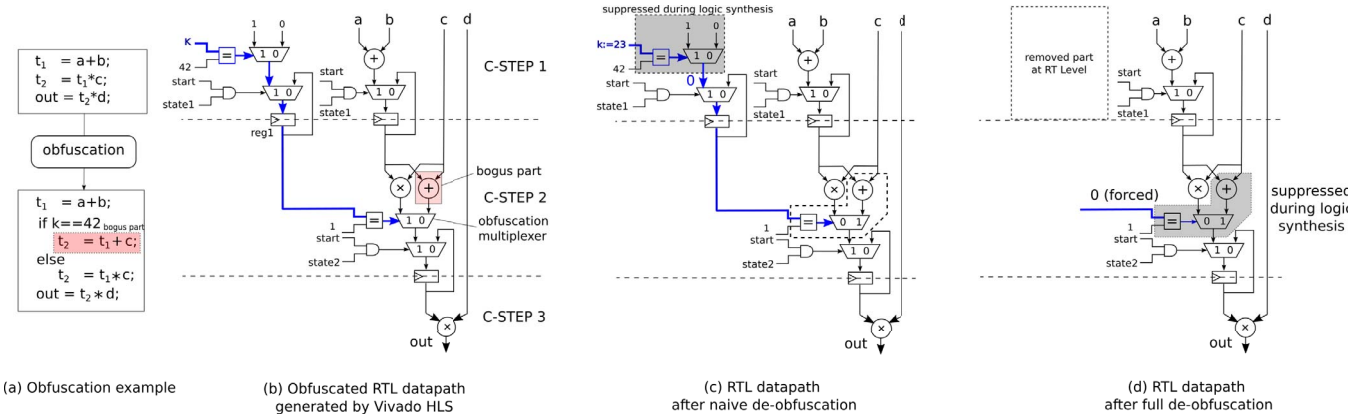

Fig. 5: Illustration of the de-obfuscation process

2) Bogus code: For this obfuscation technique, candidate AST nodes are binary expressions of the form: `<binaryExpression>::=<expression><op><expression>`, where *op* is a bitwise or arithmetic operator. For each selected binary expression, a similar bogus expression is created. The bogus binary expression contains the same expressions as the original code but a different operator. For example:

$$a = b + c \quad \Longrightarrow \quad a = b - c$$

The choice of the operator added in the bogus expression has some importance in improving security but also in reducing overhead. The chosen operator has to be computationally similar to increase stealth (a bitwise operator cannot be added for an operation between two integers for example) and of equal or lower complexity. For example, for a PLUS or a bitwise AND operator in the original expression, we respectively add a MINUS or a XOR operator in the bogus expression.

D. De-obfuscation by Key Injection

In our proposed design flow, after HLS is performed in the cloud, the design company receives obfuscated RTL code. In order to limit design overhead and to allow correct execution of the code, the code needs to be de-obfuscated (see Fig.2). At RTL level, the keys that were added as arguments to the main function in C, are now inputs of the top component. With the obfuscation method used in our framework, the RTL code has, for each obfuscated expression, a comparison between the key signal and the expected value. Depending on the result of this comparison, a multiplexer then selects which operator to use in the binary expression.

1) Naive key injection: A first approach could consist in injecting the keys externally, at the circuit interfaces, by creating a new top-level component that explicitly states the values of the obfuscation keys and injects them into the kernel.

During logic synthesis, the following behavior is expected: if the correct keys are injected, through constant propagation the predicates should evaluate correctly, and the bogus code branches should be removed by logic simplification. The circuit would then function correctly and the design overhead compared with the original, unobfuscated circuit should be close to null.

However, this procedure is only effective if the data path added by obfuscation is fully combinational. In practice, the scheduling performed during HLS results in most cases in a sequential data path. More precisely, the comparison between the key input and its expected value is often scheduled in a different clock cycle than the obfuscation multiplexer (Fig.5(b)), which controls whether to use the original or the bogus operator. In this case, the result of the key comparison is stored in a dedicated register ("reg1" on Fig.5(b)). The value of this register depends not only on the key inserted in the circuit, but also on the state of the FSM responsible for controlling the scheduling. This means that even if the result of the key comparison is fixed by injecting the keys, the value of the register is not a constant and is unknown by the logic synthesis tool. The register cannot be removed during synthesis, and the bogus code, which is tied to this register, is not removed either. The register acts as a barrier against logic simplification. While the circuit is functionally correct, the remaining bogus code (Fig.5(c)) causes a significant design overhead.

2) Proposed key injection: We propose a second approach to circumvent the previously presented issue and to de-obfuscate more thoroughly. Instead of injecting the keys externally and relying purely on logic collapsing, we locally modify the RTL code itself by performing the following operations. For each obfuscated expression, we compare the key input and its expected value. The result of this comparison is computed as 0 or 1 (0 on the example Fig.5(d)). The aforementioned barrier register thus depends on a constant (0 or 1) and the state of the controller. While logic synthesis alone cannot bypass this register, our approach enables us to know what value should command the obfuscation multiplexer. This allows to completely remove the key comparison and the associated (if any) register in the RTL code, and instead directly inject this value into the command of the obfuscation multiplexer. Because the multiplexer command is forced, the synthesis tool is able to deduce which operator must be used and thus which code branch to remove. With this approach, functionality of

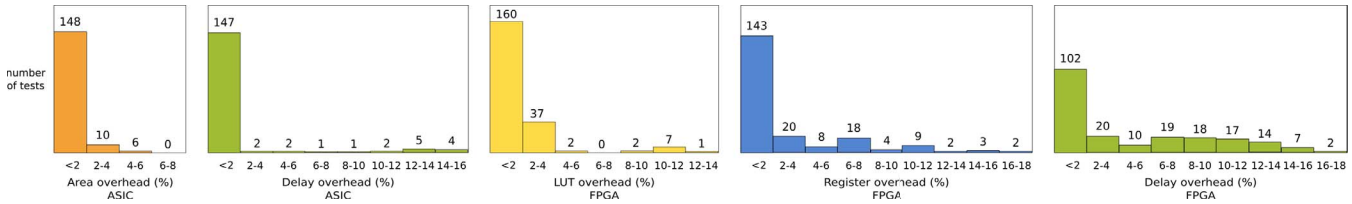

Fig. 6: Histograms of the distribution of tests by overhead

Benchmark	No De-obfuscation		Naive De-obfuscation		**Full De-obfuscation**	
	Overhead in %		Overhead in %		Overhead in %	
	Area	Delay	Area	Delay	Area	Delay
Adpcm	2.61	-0.06	2.29	-0.08	**-0.08**	**-0.01**
AES	0.3	0	0.31	0	**0.03**	**2.72**
Merge Sort	0.07	0	0.07	0	**0.03**	**0**
Mips	1.14	0.5	1.45	0.5	**-1.12**	**-7.47**
Needwun	9.01	11.69	8.39	11.69	**0.26**	**0**
Stencil3d	9.38	0.73	9.2	0.73	**1.54**	**-0.14**

(a) Average design overhead on ASIC

No De-obfuscation			Naive De-obfuscation			**Full De-obfuscation**		
Overhead in %			Overhead in %			Overhead in %		
LUT	Register	Delay	LUT	Register	Delay	LUT	Register	Delay
5,82	28,9	15,9	-0.08	-0.2	10,85	**-0.12**	**-0.2**	**11,06**
29	20	1,98	7,05	0,42	3,35	**8,97**	**0,42**	**4,49**
-1.37	18,7	2,57	-6.18	4,08	-1.27	**-5.99**	**4,08**	**-3.77**
45,9	210	7,51	1,46	-1.39	4,51	**1,78**	**-1.39**	**4,61**
23,9	33,5	-4.22	0,79	0,01	-2.78	**-0.6**	**-0.25**	**-1.98**
20,7	22,4	-9.44	1,38	4,56	-3.78	**-1.25**	**5,57**	**-0.86**

(b) Average design overhead on FPGA

Fig. 7: Average design overhead per benchmark on ASIC and FPGA

the circuit is preserved. The removal of any code added by obfuscation results in close to no overhead. This guarantees that our obfuscation process is transient.

IV. Experimental Study

A. Experimental Setup

We implemented our approach in a dedicated toolset: KaOTHIC (Key-based Obfuscating Tool for HLS in the Cloud). KaOTHIC is written in Python and uses a series of fully automated steps for obfuscation and de-obfuscation. To parse the original C files before obfuscation, an in-house compiler front-end is used. In order to evaluate our obfuscation approach, 6 benchmarks from MachSuite [15] and CHStone [16] were used: advanced encryption standard (`AES`), adaptive differential pulse code modulation encoder (`Adpcm`), optimal sequence alignment algorithm (`Needwun`), mergesort algorithm (`Merge_sort`), simplified Mips processor (`Mips`) and three-dimensional stencil computation (`Stencil3d`).

We performed a series of tests for each benchmark by obfuscating it with varying obfuscation levels from 5 to 100%. For each test, 4 logic syntheses were performed: the original design, the obfuscated design, the naively de-obfuscated design and the fully de-obfuscated design. This allowed us to compare design overhead of the different approaches.

For HLS we used Xilinx Vivado HLS. Logic synthesis was performed with Synopsys Design Compiler, using the Synopsys SAED 90nm educational library for ASIC, as well as with Xilinx Vivado for a Nexys 4 DDR Artix-7 FPGA board. It is important to note that DSP blocks were disabled for FPGA logic synthesis in order to restrict our surface analysis to LUT and register count.

B. Results and analysis

Using the previously presented method and KaOTHIC tool, we performed a total of 164 tests on ASIC and FPGA.

The average runtime overhead for HLS is negligible (under 5%). In over 95% of our tests, the area overhead is below 5%, which we consider an acceptable overhead threshold. On ASIC, the delay overhead is below 5% in 91% of the tests. In respectively 80% and 60% of the tests, the register and delay overhead on FPGA are also below 5%.

Fig.6 represents the distribution of the 164 test results in terms of area and delay overhead for ASIC, and in terms of LUT, register and delay overhead for FPGA. Additionally, the tables on Fig. 7 show the detailed average results for each benchmark, without de-obfuscation and with both de-obfuscation approaches. The results indicate that design overhead is high when no de-obfuscation is performed. When using a naive de-obfuscation approach, overhead slightly decreases. Finally, overhead can be strongly reduced by smartly de-obfuscating the circuits through our proposed key injection approach. This proves that we are successfully able to remove the code added during obfuscation. To ensure the correctness of our obfuscating method, i.e. to verify that the obfuscated code has the same functionality as the original code, we simulated the code at RTL level using GHDL, an open-source VHDL compiler. Simulation results show that the original code without obfuscation and the obfuscated code after correct key injection have the same behavior, while obfuscated code with wrong keys injected does not behave correctly. Our approach is thus able to safely remove the additional code without jeopardizing the functionality of the circuit.

In some test cases (e.g. Mips on ASIC), we obtained a negative overhead. This may happen when the increased code size forces the HLS tool to make different optimization choices, thus sometimes resulting in a smaller design than the original one. The small remaining overhead (e.g. Stencil3D) is in our opinion due to resource sharing and to the impact that the syntax variations created by the obfuscation process have on HLS tools. On FPGAs, overheads may vary more importantly, while remaining acceptable, because of the components (LUTs, registers, interconnects...) used to make these devices reconfigurable and the associated logic synthesis techniques.

Further work will focus on better mastering overheads to offer a quality of results similar to the ASIC ones.

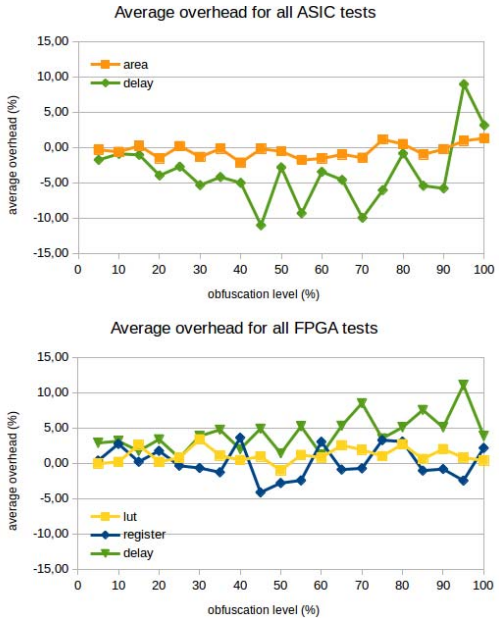

Fig. 8: Average overhead per obfuscation level for all performed tests

Through a series of tests done with obfuscation levels ranging from 5% to 100%, we studied correlation between overhead and obfuscation level. Our results indicate that there is no correlation between obfuscation level and area or delay overhead, as shown on Fig.8. We can for example note that area overhead varies between approximately -2% and 2%, and does not increase with obfuscation level. These results demonstrate that the maximum level of obfuscation, i.e. 100%, can always be chosen.

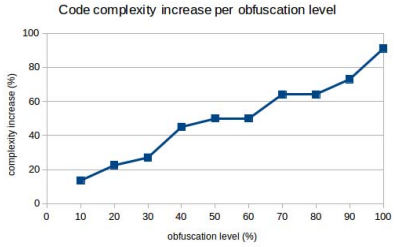

Fig. 9: Cyclomatic complexity increase of Needwun source code after obfuscation per obfuscation level

To evaluate the security of the proposed obfuscation technique, we calculated the cyclomatic complexity. While not a precise indicator of the difficulty of attack, a higher complexity does usually imply that the code is more difficult to understand. Our results show that after obfuscation, the complexity increases on average by 109%. Furthermore, the increase in complexity is positively correlated with the level of obfuscation, see Fig.9 for an example with Needwun benchmark, which is representative of the results obtained for other benchmarks. Increasing the level of obfuscation can thus be a simple way to improve security without incurring a significant increase in overhead.

V. Conclusions and Future Work

In this work, we proposed using obfuscation methods to secure BIPs during HLS in a cloud context. Experimental results show that we are able to obfuscate using a key-based technique and to de-obfuscate safely after HLS with minimal overhead. In the future, the very positive first results on ASIC and FPGA encourage us to continue exploring this approach both by trying to optimize and further reducing overall overhead, and by diversifying the obfuscation techniques to achieve a higher level of security. We also plan to work on a more detailed evaluation of security.

References

[1] B. Bailey, "EDA In The Cloud", *https://semiengineering.com/eda-in-the-cloud/*, April 2018, last accessed: 08/09/18

[2] C. Collberg, C. Thomborson and D. Low, "A taxonomy of obfuscating transformations", in *Technical Report 148, Department of Computer Science, University of Auckland*, July 1997

[3] D. Kirovski, Y.-Y. Hwang, M. Potkonjak and J. Cong, "Intellectual Property Protection by Watermarking Combinational Logic Synthesis Solutions", in *Proceedings of the 1998 IEEE/ACM International Conference on Computer-aided Design*, November 1998

[4] M. Lewandowski, R. Meana, M. Morrison and S. Katkoori, "A Novel Method for Watermarking Sequential Circuits", in *IEEE International Symposium on Hardware-Oriented Security and Trust*, June 2012

[5] S. Garg and J.J.V. Rajendran, "Split Manufacturing", in *Hardware Protection through Obfuscation, Springer International Publishing*, pp. 243-262, 2017

[6] R.S. Chakraborty and S. Bhunia, "RTL Hardware IP Protection Using Key-Based Control and Data Flow Obfuscation", in *2010 23rd International Conference on VLSI Design*, January 2010

[7] Y. Lao and K.K. Parhi, "Obfuscating DSP Circuits via High-Level Transformations", in *IEEE Transactions on Very Large Scale Integration Systems 23*, May 2015

[8] A. Sengupta, D. Roy, S. Mohanty and P. Corcoran, "DSP Design Protection in CE through Algorithmic Transformation Based Structural Obfuscation", in *IEEE Transactions on Consumer Electronics*, November 2017

[9] S. A. Islam and S. Katkoori, "High-Level Synthesis of Key Based Obfuscated RTL Datapaths", in *19th Int'l Symposium on Quality Electronic Design*, March 2018

[10] C. Pilato, F. Regazzoni, R. Karri and S. Garg, "TAO: Techniques for Algorithm-Level Obfuscation during High-Level Synthesis", in *Design Automation Conference*, June 2018

[11] N. Veeranna and B. C. Schafer, "Efficient Behavioral Intellectual Properties Source Code Obfuscation for High-Level Synthesis", in *2017 18th IEEE Latin American Test Symposium*, March 2017

[12] H. Xu, Y. Zhou, Y. Kang and M. R. Lyu, "On Secure and Usable Program Obfuscation: A Survey", *ArXiv e-prints*, October 2017

[13] D. Apon , Y. Huang, J. Katz, and A. J. Malozemoff, "Implementing cryptographic program obfuscation", *Cryptology ePrint Archive, Report 2014/779*, 2014

[14] C. Collberg, C. Thomborson and D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", in *Symposium on Principles of Programming Languages*, 1998

[15] B. reagen, R. Adolf, Y.S. Shao, G. Wei and D. Brooks, "MachSuite: Benchmarks for Accelerator Design and Customized Architectures", in *Proceedings of the IEEE International Symposium on Workload Characterization*, October 2014

[16] Y. Hara, H. Tomiyama, S. Honda and H. Takada, "Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis", in *Journal of Information Processing*, Vol.17, pp.242-254, 2009