Invited: High-level design methods for hardware security: Is it the right choice?

Christian Pilato
Donatella Sciuto
Politecnico di Milano, Italy

Benjamin Tan
University of Calgary, Canada

Siddharth Garg
Ramesh Karri
New York University, USA

ABSTRACT

Due to the globalization of the electronics supply chain, hardware engineers are increasingly interested in modifying their chip designs to protect their intellectual property (IP) or the privacy of the final users. However, the integration of state-of-the-art solutions for hardware and hardware-assisted security is not fully automated, requiring the amendment of stable tools and industrial toolchains. This significantly limits the application in industrial designs, potentially affecting the security of the resulting chips. We discuss how existing solutions can be adapted to implement security features at higher levels of abstractions (during high-level synthesis or directly at the register-transfer level) and complement current industrial design and verification flows. Our modular framework allows designers to compose these solutions and create additional protection layers.

1 INTRODUCTION

The cost of chip manufacturing is increasing exponentially as the technology scales down. Only a few companies can afford such costs, forcing many semiconductor design houses to become *fab-less.* A fab-less design house has the core of its business in the design of the integrated circuit, outsourcing the manufacturing step to a third-party company. On the one hand, this process is cost-efficient and allows the co-existence of many semiconductor players along with only a few foundries. On the other hand, it opens up several security concerns in the semiconductor supply chain, as shown in Figure 1. For example, malicious foundry employees have access to the design files and can reverse engineer the functionality to create illegal copies (also with the help of activated chips coming from the market) [13]. So, many design houses are integrating security countermeasures in their designs at different abstraction levels [4].

To hide the chip functionality, designers can use two classes of approaches: logic locking and (e)FPGA redaction. In the former, the designer adds extra logic, controlled by a new input (*key*), such that the circuit behaves correctly only when the correct input key is provided [13]. In the latter, parts of the circuit are replaced by embedded FPGAs such that the foundry has no clue about the functionality that will be implemented on the reconfigurable logic [2].

Permission to make digital or hard copies of all or part 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 components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9142-9/22/07...$15.00
https://doi.org/10.1145/3489517.3530635

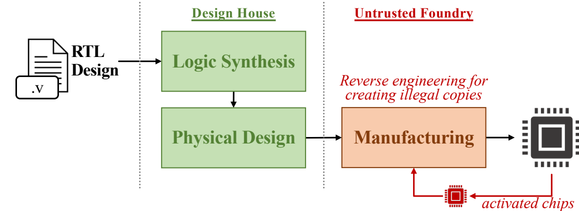


Figure 1: Security threats in the semiconductor supply chain.

While these approaches have several advantages, designers need to face multiple challenges to integrate them into EDA flows. First, the protection of the circuit must be semantically meaningful. Gate-level locking can only protect the structure of the circuit after it has been already synthesized (and optimized) with logic synthesis. Raising the abstraction level is thus to operate on and protect the circuit behavior rather than its structure [9]. Second, attackers have developed many attacks for *breaking* the protection (i.e., recovering the locking key or the eFPGA bitstream), including machine learning-based [14] and SAT-based attacks [15]. Such protections must be in continuous evolution to make the life of the attackers exponentially harder. Third, adding the logic for logic locking or the extra eFPGAs introduces significant overhead to the chip. To limit such overhead, designers need to carefully select where to add the extra logic or decide how many and which eFPGA should be added without compromising the security of the design [7].

In this paper, we present an **EDA flow to integrate high-level solutions for hardware IP protection**. In particular, we operate at the register-transfer level to better identify the semantics to be protected before further optimizations. Our proposed flow combines *RTL locking* and *FPGA redaction* to create a single solution that is compatible with standard industrial design flows. Our main contributions are:

- we propose a unified design flow that partitions the design, applies the proper countermeasures, recreates a final description ready for synthesis;
- we integrate state-of-the-art solutions for RTL locking and FPGA customization to protect selected parts of the design;
- we discuss the major challenges for this approach.

Our approach can enable further research at the system level, broadening the spectrum of solutions that can be explored and evaluated.

2 BACKGROUND

2.1 Threat Model

In this work, we aim at protecting the hardware intellectual property (IP) against an untrusted foundry. The attackers have access to the IC's layout files (GDSII) from which they can perform reverse engineering to extract the corresponding RTL [11, 12]. They can

also perform simulations on the resulting RTL to get information about the output results. They may have access to a working chip (*oracle*), which can be obtained by the (black) market, to study I/O relationships and help key recovery.

2.2 Logic Locking

Logic locking is a technique to hide and alter the functionality of a circuit. During the design phase, the circuit is modified to insert additional logic that is controlled by a new input, called the *locking key*, which is known to the designer but not given to the foundry. In this way, even if the circuit is reverse engineered, the correct functionality cannot be reproduced without having the right key.

Logic locking can be applied at different abstraction levels, including at the specification level [8], during high-level synthesis [10], at the register-transfer level [9], and directly on gate-level netlists. The general idea is that, with a key of K bits, the probability of guessing the correct functionality (without any additional information) is 2^{-K}. So, designers must define their locking protections so all alternatives are equally plausible.

However, logic locking solutions have been widely studied and attacked. When the attackers have only the locked description, they can analyze it to identify some functional or structural bias and use it to get information about the key. Functional bias can be identified in the probability distribution of the outputs. For example, a trivial locking schema can always emit 0 on the outputs except when the correct key is provided. Even if the circuit behaves correctly only with the right key, this can be clearly retrieved by the attacker. Designers must corrupt the output with a uniform probability of getting 0 or 1 so that, even with a brute force attack, no information is available on the key. Structural attacks include re-synthesis and machine learning-based to identify (and eliminate) the extra logic [14]. When the attackers have an oracle, they can study the correct input/output relationships. To do so, they can formulate the key recovery problem as an SAT problem [15] to identify *distinguishable input patterns* and rule out incorrect keys.

2.3 FPGA Redaction

Embedded FPGAs are reprogrammable devices that can be integrated into integrated circuits to hide the entire functionality of selected modules. Protection is given by the reprogrammability feature of such components. During manufacturing, the FPGA does not contain any information about the functionality that can be implemented. It will be inserted only later by uploading the proper *bitstream*, i.e., the FPGA configuration file. So an attacker must recover the entire bitstream to replicate the correct functionality. Conversely, an FPGA can load any *valid bitstream*[1], which does not necessarily implement the original functionality.

Recent studies have shown that the security of eFPGA redaction is more related to the parameters of the eFPGA instance that is added to the design rather than the modules that are implemented on it [1, 2]. However, the designers have to maximize the utilization of the reconfigurable logic and the I/O pins. Hence, using FPGA customization tools [5, 16] is a viable solution for creating tailored FPGAs that implement the redacted modules.

[1]A valid bitstream is an FPGA configuration that creates an admissible functionality between input and output pins.

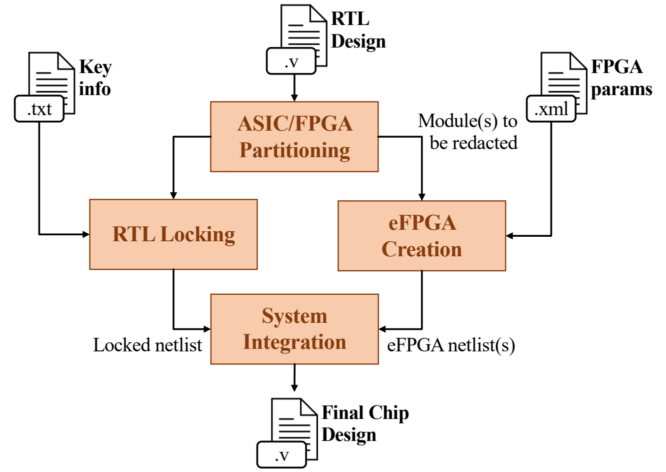


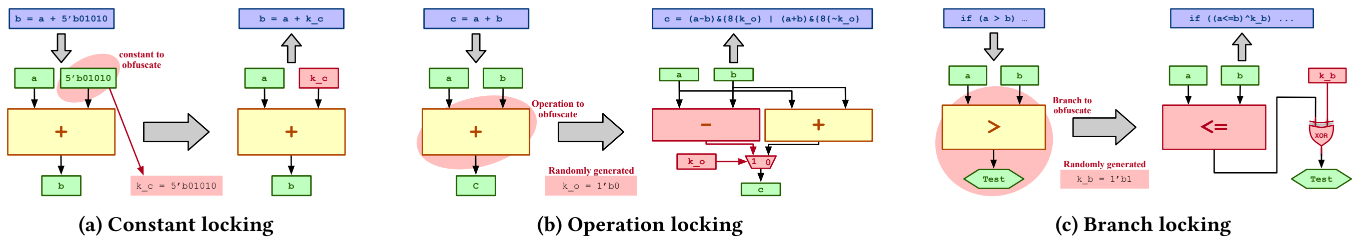
Figure 2: Design framework to apply hardware IP protection at high abstraction levels

3 HOW TO RAISE THE ABSTRACTION LEVEL FOR HARDWARE IP PROTECTION

To protect the hardware IP against an untrusted foundry, we propose a framework that allows the designers to **combine several system-level protection techniques**, including RTL locking and FPGA redaction, in an industrial design flow. Our framework, which is shown in Figure 2, is a pre-processing step before logic synthesis (see Figure 1). It starts from an RTL description of the chip to be fabricated and some designers' parameters (e.g., the information about the locking key, the eFPGA parameters, and the most critical outputs to be "protected"). The framework produces the implementation files of the chip, i.e., a synthesizable RTL description of the circuit that includes both the ASIC part and the eFPGA netlists, ready for the subsequent logic and physical synthesis steps. The framework is composed of the following steps:

(1) **ASIC/FPGA partitioning**: it determines which modules will be implemented directly in ASIC and which ones will be implemented on the reconfigurable logic of the eFPGAs.
(2) **RTL locking**: for the parts to be implemented directly in ASIC, it applies RTL locking to hide their semantics.
(3) **eFPGA creation**: for the parts to be mapped onto reconfigurable logic, it customizes the eFPGA instances to be included in the chip.
(4) **System integration**: the two descriptions are merged into a unique one to be passed to logic synthesis and physical design steps. This step also manages the additional logic to provide (and protect) the locking key or to upload the configuration bitstream.

In the following, we detail all these parts. We show how it is possible to integrate state-of-the-art techniques that operate at higher levels of abstraction. All techniques operate on hardware descriptions, so it is possible to use industrial formal verification methods to certify that the chip behaves correctly if and only if the correct "secrets" (i.e., locking key and FPGA configuration bitstreams) are provided.



(a) Constant locking (b) Operation locking (c) Branch locking

Figure 3: Three ASSURE locking techniques [9]: (a) constants, (b) operations, and (c) branches.

3.1 ASIC/FPGA Partitioning

This phase receives the RTL description of the chip to be fabricated and the design constraints. It aims at identifying the blocks that are more *critical* for the selected outputs. Indeed, not all inputs and outputs are equally important from the security viewpoint. For example, the external interfaces may implement standard protocols (like AXI4 transactions) and they do not necessitate IP protection. Conversely, the data outputs of digital filters may reveal proprietary information on its response.

The framework first lists all modules of the designer and initializes an initial score to zero for each of them. A “bonus score” is also assigned to each selected output. In this way, it is possible to differentiate them. Then, the framework builds a *system dependence graph* that captures the I/O relationships of the design. This representation is used to identify which modules have a direct effect on each of the selected outputs. Indeed, let b_i the bonus score associated with the output i. The score of each module that has an effect on the output i will be increased by b_i. After this procedure, all modules are ranked based on the resulting scores and the top ones are selected for eFPGA redaction. The top-score modules may be the ones that have an effect on multiple selected outputs (i.e., they received more bonus scores) or have an effect on more critical outputs (i.e., they received higher bonus scores). The number of modules to be selected is defined by the user and can be further refined by additional constraints (e.g., the maximum number of input and output pins).

The selected modules are forwarded to the eFPGA creation step, while the ones will be implemented directly in ASIC, possibly with an additional level of protection given by RTL locking.

3.2 RTL Locking

Our RTL locking flow operates directly on synthesizable RTL descriptions. In this way, it fits within existing EDA flows and the same constraints as the original design (e.g., clock period, pin assignments, etc.) Designers can also use formal verification tools to check whether the resulting RTL is equivalent to the original design if and only if the correct key is used. We apply the following state-of-the-art RTL locking techniques [9]:

(1) **Constant locking**: We protect confidential constants by replacing them entirely with key bits carrying constant information. For example, we rewrite the RTL operation $out = in + 8\text{'b}11101001$ as $out = in + K_c$ where K_c is the 8-bit constant stored in the locking key. The attacker has 2^8 possibilities to get a correct *out* for a given *in* value.
(2) **Operation locking**: We introduce a multiplexer to decide between the correct operation and a dummy operation based on the value of one key bit. For example, we lock the operation $out = in_1 + in_2$ as $out = K_o\ ?\ (in_1 + in_2) : (in_1 - in_2)$. Only the correct value of K_o enables the propagation of the correct result.
(3) **Branch locking**: In branch obfuscation, we perform XOR based locking on the original condition followed by logical inversion if key-bit is 1. For e.g., we rewrite the original condition $in_1 > in_2$ as $(in_1 \leq in_2) \oplus K_b$ with locking key-bit $K_b = 1$. An attacker cannot deduce from the statement whether the original condition was $>$ or $\leq$.

The application of the three techniques is shown in Figure 3. The standard flow operates by locking the elements in topological order until the maximum number of available key bits is reached. The approach can be implemented by a depth-first visit of the abstract syntax tree (AST), which is directly extracted by parsing the RTL description. After modifying the AST, the corresponding RTL is generated. The resulting design has the same external interface as the original module, except for an additional input port that is connected to the place where the locking key will be stored (see Section 3.4). The key is composed of two parts: one contains the constant information extracted from the design and the other contains the key bits for locking operations and branches.

The same approach can be extended with methods for better selecting the locking points [3] or with additional logic to protect the locking key against SAT attacks [6]. These methods provide additional protections against a wide range of attacks. For example, the same system dependence graph used for ASIC/FPGA partitioning can be reused to identify the effects of each RTL statement on the outputs, preferring the ones that can lead to more corruptibility [3]. When the designers aim at protecting the design against oracle-based attacks, they can “isolate” a selected portion of the locking key in test mode so that even the activated chip is not able to guarantee correct outputs, making the oracle *untrustable*.

3.3 eFPGA Creation

Once the designers have selected modules to be redacted, it is necessary to create the corresponding eFPGA instances. For doing this, we use open-source FPGA customization frameworks (like OpenFPGA [16] or FABulous [5]). For example, OpenFPGA can start from the description of one or more HDL modules and the fabric parameters, and create the netlist (and the associated bitstream) of the corresponding eFPGA instance. An example of the OpenFPGA flow is shown in Figure 4. This FPGA instance contains the minimum

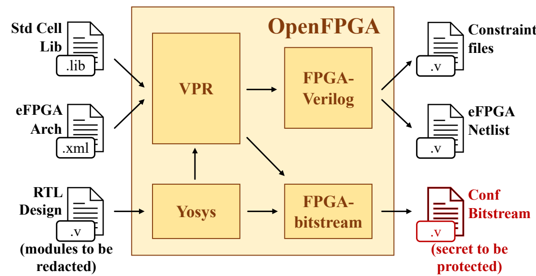


Figure 4: Our eFPGA redaction flow based on OpenFPGA for FPGA customization [2, 7].

number of configurable blocks that can implement the given module(s). Using FPGA customization tools allow us to create instances that (1) have less overhead than off-the-shelf FPGAs and (2) have maximum utilization of both I/O pins and configurable logic blocks, which guarantees high resilience to SAT attacks [1, 2].

3.4 System Integration

The last step of our flow requires integrating the two parts (ASIC and eFPGAs) to create a single hardware description for logic synthesis and physical design, adding the extra logic for interfacing the two parts and managing the locking key.

In this phase, there are two major EDA issues: (1) the designers need to remap the in/out pins of the newly-created eFPGA onto the original wires to ensure functional correctness, and (2) ASIC and eFPGA may run at different clock frequencies and they must be properly interfaced to obtain a working chip.

For the functional integration of the eFPGA instances, the designers need to identify the wires that were connecting the design with the modules that have been replaced with eFPGAs. This process can be automated thanks to the artifacts produced by the FPGA customization tools and FPGA pins that are not remapped to original wires are set to 0. These pins are clearly associated with reconfigurable logic that is not relevant to the redacted design, reducing the number of configuration bits that must be recovered by the attackers and so reducing the overall security of the design. This is why it is important to maximize the utilization of the FPGA pins.

From the EDA viewpoint, integrating the eFPGA instances requires evaluating the critical paths introduced by the reconfigurable logic. To avoid slowing down the entire chip, the designers may isolate the FPGA instances and run time with a different clock frequency. To create a design with multiple frequency domains, designers may need to introduce *synchronizers* at the boundaries of each domain. On the one hand, this can solve the integration issue but it may introduce time/area overhead in the design. So, this process has implications for the selection of the modules to be redacted. The back-annotation of this information for further refining the selection is an ongoing research effort.

4 DISCUSSION AND CONCLUSIONS

In this work, we presented a framework to raise the abstraction level of hardware IP protection to the register-transfer level. Working at RTL enables the integration into established industrial design flows and, at the same time, combine several techniques for protecting the semantics of the circuit more efficiently. In particular, we partition the given design into two parts: one to be implemented directly in ASIC and the other to be programmed onto an embedded FPGA that is added to the chip design. While the former is more cost-efficient, it can be reverse-engineered and so we apply RTL locking for its protection. The latter part leverages the reprogrammability features of FPGA devices to completely hide the functionality during manufacturing. However, the integration of eFPGAs is still complex and expensive. So, the correct partitioning is still an open issue to trade-off security and hardware cost. To answer the question that opened this paper, we strongly believe that high-level methods are the right choice for hardware IP protection. Indeed, **reasoning at higher levels of abstraction allows designers to explore more alternatives and restructure the chip's design accordingly to the security metrics.** Also, designers can better analyze the designs, identify more easily the semantics to be protected, and apply stronger and cost-effective protections.

ACKNOWLEDGMENTS

The authors would like to thank Jitendra Bhandari, Abdul Khader Thalakkattu Moosa, and Luca Collini (NYU) for their support in this research. R. Karri was supported in part by ONR Award # N00014-18-1-2058, NSF Grant # 1526405, NYU Center for Cybersecurity, and NYUAD Center for Cybersecurity.

REFERENCES

[1] J. Bhandari, A. K. T. Moosa, B. Tan, C. Pilato, G. Gore, X. Tang, S. Temple, P.-E. Gaillardon, and R. Karri. 2021. Not All Fabrics Are Created Equal: Exploring eFPGA Parameters For IP Redaction. arXiv:2111.04222
[2] J. Bhandari, A. K. Thalakkattu Moosa, B. Tan, C. Pilato, G. Gore, X. Tang, S. Temple, P.-E. Gaillardon, and R. Karri. 2021. Exploring eFPGA-based Redaction for IP Protection. In *IEEE/ACM ICCAD*.
[3] L. Collini, R. Karri, and C. Pilato. 2022. A Composable Design Space Exploration Framework to Optimize Behavioral Locking. *ACM/IEEE DATE* (2022).
[4] W. Hu, C.-H. Chang, A. Sengupta, S. Bhunia, R. Kastner, and H. Li. 2021. An Overview of Hardware Security and Trust: Threats, Countermeasures, and Design Tools. *IEEE Transactions on CAD of Integrated Circuits and Systems* 40, 6 (2021).
[5] D. Koch, N. Dao, B. Healy, J. Yu, and A. Attwood. [n. d.]. FABulous: An Embedded FPGA Framework. In *ACM/SIGDA FPGA*. 45–56.
[6] N. Limaye, A. B. Chowdhury, C. Pilato, M. T. M. Nabeel, O. Sinanoglu, S. Garg, and R. Karri. 2021. Fortifying RTL Locking Against Oracle-Less (Untrusted Foundry) and Oracle-Guided Attacks. *ACM/IEEE DAC* (2021).
[7] C. Muscari Tomajoli, L. Collini, J. Bhandari, A. K. Thalakkattu Moosa, B. Tan, X. Tang, , P.-E. Gaillardon, R. Karri, and C. Pilato. 2022. ALICE: An Automatic Design Flow for eFPGA Redaction. In *IEEE/ACM DAC*.
[8] M. R. Muttaki, R. Mohammadivojdan, M. Tehranipoor, and F. Farahmandi. 2021. HLock: Locking IPs at the High-Level Language. In *ACM/IEEE DAC*. 79–84.
[9] C. Pilato, A. B. Chowdhury, D. Sciuto, S. Garg, and R. Karri. 2021. ASSURE: RTL Locking Against an Untrusted Foundry. *IEEE Trans. on VLSI Systems* 29, 7 (2021).
[10] C. Pilato, F. Regazzoni, R. Karri, and S. Garg. 2018. TAO: Techniques for Algorithm-Level Obfuscation during High-Level Synthesis. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.
[11] R. S. Rajarathnam, Y. Lin, Y. Jin, and D. Z. Pan. 2020. ReGDS: A Reverse Engineering Framework from GDSII to Gate-level Netlist. In *IEEE HOST*. 154–163.
[12] J. Rajendran, A. Ali, O. Sinanoglu, and R. Karri. 2015. Belling the CAD: Toward Security-Centric Electronic System Design. *IEEE Transactions on CAD of Integrated Circuits and Systems* 34, 11 (Nov. 2015), 1756–1769.
[13] K. Shamsi, M. Li, K. Plaks, S. Fazzari, D. Z. Pan, and Y. Jin. 2019. IP Protection and Supply Chain Security through Logic Obfuscation: A Systematic Overview. *ACM Trans. on Design Automation of Electronic Systems (TODAES)* 24, 6 (2019), 1–36.
[14] D. Sisejkovic, L. M. Reimann, E. Moussavi, F. Merchant, and R. Leupers. 2021. Logic Locking at the Frontiers of Machine Learning: A Survey on Developments and Opportunities. In *IFIP/IEEE VLSI-SoC*. 1–6. https://doi.org/10.1109/VLSI-SoC53125.2021.9606979
[15] P. Subramanyan, S. Ray, and S. Malik. 2015. Evaluating the security of logic encryption algorithms. In *IEEE HOST*. 137–143.
[16] X. Tang, E. Giacomin, B. Chauviere, A. Alacchi, and P.-E. Gaillardon. 2020. OpenFPGA: An Open-Source Framework for Agile Prototyping Customizable FPGAs. *IEEE Micro* 40, 4 (2020), 41–48.