Highway to HAL

Open-Sourcing the First Extendable Gate-Level Netlist Reverse Engineering Framework

Sebastian Wallat*, Nils Albartus, Steffen Becker, Max Hoffmann, Maik Ender, Marc Fyrbiak, Adrian Drees, Sebastian Maaßen, Christof Paar*
Ruhr-Universität Bochum
Horst Görtz Institut für IT-Security
Bochum, Germany
{firstname.lastname}@rub.de

ABSTRACT

Since hardware oftentimes serves as the root of trust in our modern interconnected world, malicious hardware manipulations constitute a ubiquitous threat in the context of the Internet of Things (IoT). Hardware reverse engineering is a prevalent technique to detect such manipulations.

Over the last years, an active research community has significantly advanced the field of hardware reverse engineering. Notably, many open research questions regarding the extraction of functionally correct netlists from Field Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs) have been tackled. In order to facilitate further analysis of recovered netlists, a software framework is required, serving as the foundation for specialized algorithms. Currently, no such framework is publicly available.

Therefore, we provide the first open-source gate-library agnostic framework for gate-level netlist analysis. In this positional paper, we demonstrate the workflow of our modular framework HAL on the basis of two case studies and provide profound insights on its technical foundations.

KEYWORDS

hardware reverse engineering, gate-level netlist, open-source framework

ACM Reference Format:
Sebastian Wallat*, Nils Albartus, Steffen Becker, Max Hoffmann, Maik Ender, Marc Fyrbiak, Adrian Drees, Sebastian Maaßen, Christof Paar. 2019. Highway to HAL: Open-Sourcing the First Extendable Gate-Level Netlist Reverse Engineering Framework. In *Proceedings of Malicious Software and Hardware in Internet of Things (MAL-IOT19).* ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3310273.3323419

*Sebastian Wallat and Christof Paar are also affiliated with University of Massachussets, Amherst, USA

CF '19, April 30-May 2, 2019, Alghero, Italy

ACM ISBN 978-1-4503-6685-4/19/05.
https://doi.org/10.1145/3310273.3323419

1 INTRODUCTION

In an increasingly interconnected world, hardware components serve as the root of trust in virtually any computing system. Therefore malicious hardware manipulations of mission-critical components can have serious implications, ranging from simple revenue loss over faults in critical infrastructure up to life-threatening consequences [8].

On the one hand hardware reverse engineering is the tool-of-choice to identify such manipulations, and to check the trustworthiness of hardware in general [3, 8]. It encompasses the detection of potentially harmful counterfeits and copyright infringements. On the other hand, hardware reverse engineering is often used to insert hardware Trojans [28], which weakens the security of a system or to commit said copyright infringements by trying to counterfeit or simply copy intellectual property.

Since hardware reverse engineering is a highly complex process, semi-automated tools are desperately needed by the community [31].

In the world of software reverse engineering comprehensive and expandable frameworks covering the complete workflow of binary analysis exist, e.g., IDA Pro or Ghidra. However, for hardware reversing there is no such framework yet [27] but loose collections of scripts, e.g., [13]. To the best of our knowledge, we are the first to release a fully customizable open source gate-level netlist reverse engineering framework [9] to the open-source community via GitHub[1].

We encourage the community to conduct their own research using our framework in the field, to write and publish plugins for specialized reverse engineering tasks, and to support the development of HAL as a whole. To get the interested parties started, we provide guidance for the rich feature set and its technical foundations for both HAL users and developers. Furthermore, we demonstrate the usage of HAL by means of two case studies: Reverse engineering Finite State Machines (FSMs), and finding watermarks.

2 BACKGROUND

In the following section we provide the essential background information for the topic of chip-level reverse engineering and introduce the field of gate-level netlist reverse engineering.

[1] HAL, https://github.com/emsec/hal

2.1 Chip-Level Reverse Engineering

With chip-level reverse engineering, an attacker extracts a human-readable gate-level netlist from the examined Integrated Circuit (IC) or FPGA.

During this phase no functional analysis of the netlist takes place. Only in the gate-level netlist reverse engineering step conducted later, the attacker analyzes the chip's logical functionality.

FPGA Reverse Engineering. Due to its volatile nature SRAM-based FPGAs are reconfigured on every boot-up by an externally stored bitstream. The bitstream contains the configuration of the basic FPGA elements, i.e., which Boolean function is evaluated in a Look-up table (LUT) and how these logical functions are connected via the routing. For reversing an FPGA bitstream, an attacker has to (i) extract the bitstream from the external memory, (ii) reverse the bitstream file format, and (iii) convert the downloaded bitstream to a human-readable netlist.

For retrieving the bitstream the attacker can either wiretap the configuration lines on the PCB, or directly read out the flash memory. Even if the attacker encounters an encrypted bitstream the chances of recovery are high as shown in [14–17, 24]. The bitstream file format reversing process has been described in recent papers [2, 5, 7, 11, 18–21, 25, 32]. All these works use the correlation method. Here, the attacker creates a basic design containing an instantiation of the examined FPGA element, e.g., a LUT, Flip Flop (FF), or the routing. In the next step, the attacker varies the elements' configuration and creates one bitstream from the basic design and one from the modified design. The difference between both bitstreams correlates to the introduced changes in the altered design. Using the correlation method the attacker can build a database of bitstream bits and their corresponding configuration in the netlist. Using this database, the attacker can convert the bitstream under attack to a human-readable netlist.

IC Reverse Engineering. In contrast to FPGAs, reversing ICs requires several steps and is considerably more complex due to shrinking technology sizes [8, 12, 22, 27]. The reversing steps consist of (i) decapsulating, (ii) delayering, (iii) image acquisition, and (iv) image processing in order to generate the human-readable netlist.

First, the IC is decapsulated mostly using wet or dry chemistry to remove the organic package material or by using mechanical means. The chemicals can fully remove the packaging, while not damaging the silicon die. In the next step, the chip is delayered and images of each layer are acquired. This step depends on the used manufacturing technologies, thus there exist several delayering techniques. On today's feature sizes, the first passivation layer is often removed with dry anisotropic etching. The next metal layers are removed via plasma etching or ion milling. The difficulties are the over-etching – especially of the die's edges – or warpages due to the mechanical stress between the substrate and the metal layer. Each of these layers are digitized via a Scanning Electron Microscope (SEM) or a Focused Ion Beam (FIB). The remaining metal layers and oxide layer is then removed with diamond suspension and dry chemistry. Using fluoric acid the active regions of the chip are revealed.

After acquiring all images from each layer the images are stitched together. Here, precise alignment is crucial to introduce no faulty transitions between two neighboring images. Finally, software assisted image processing generates the human-readable netlist by identifying standard cells first and reconstructing the connections in the metal layer second.

2.2 Gate-level Netlist Reverse Engineering

A gate-level netlist is a representation of a set of logic gates from a particular gate library together with their interconnections [29]. Combinational logic is usually implemented with Boolean gates or Look-up tables (LUTs) and multiplexers, while sequential logic is realized through Flip Flops (FFs) or latches. All these elements are defined by the regarding gate library. Netlists can either be represented textually via HDL or as a graph, where the edges depict connections and the nodes represent gates.

The absence of (1) meaningful descriptive labels, (2) boundaries of implemented modules, and (3) module hierarchies in flat gate-level netlists drastically complicates the process of gate-level netlist reverse engineering [8].

However, the representation as a graph facilitates the application of graph-based algorithms, which can help identifying the control logic or restoring certain module boundaries and hierarchies. A further approach consists in the detection of unique (logical) structures in the netlist.

3 HAL – THE HARDWARE ANALYZER

HAL aids analysts with a rich feature set to facilitate explorative functionality recovery of gate-level netlists in a semi-automated fashion. To this end HAL processes netlists in its own graph-based representation (cf. Section 2.2). Note that HAL itself is gate-library agnostic, hence it can be used to analyze netlists of ASICs as well as FPGAs.

Figure 1 shows a high-level overview of the main workflow and core features of HAL. An analyst can either parse a new netlist into HAL or continue previous work by loading a snapshot file. After HAL has loaded the netlist, the analyst can use HAL's graph core to freely traverse or even manipulate the netlist. This can be done in an explorative manner in the GUI, either via the Python shell or by direct interaction with the graph view. To perform time-critical algorithms without the performance penalties of Python, custom C++ plugins can be used, even via the Python shell. All actions performed are documented in log files and plugins can access their own logging channels to allow for straightforward report filtering. Changes to the graph, regardless of the origin, are directly reflected in the GUI elements, allowing the analyst to not only logically but also visually partition the netlist. At any point in time, the analyst can create a snapshot of the current graph representation, that can be used to resume analysis later or revert to an earlier state.

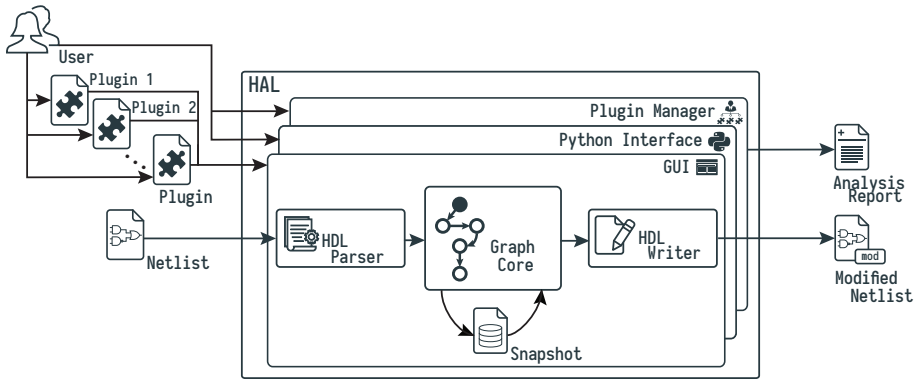


Figure 1: Overview of the original HAL architecture from [9] (modified)

Note that the GUI is entirely optional. Hence, HAL can be executed as a stand-alone command line tool, offering its full range of features except for visualization. After performing modifications, the analyst can choose to write the netlist back into a an HDL format, resulting in a synthesizable gate-level netlist.

Open Source Release. Due to the growing demand from the scientific community we decided to publicly release HAL. The source code is available on GitHub (cf. Section 1) under the open-source MIT license. We hope that HAL will be of use to the research community and encourage interested developers to contribute to the project via GitHub. We support both Linux and macOS as the Operating System (OS).

Technical Foundation

Throughout the development of HAL various aspects regarding the performance, usability, and modular expandability had to be considered. The following section highlights the emerged issues and presents our solutions.

The Core System. Since complex gate-level netlists are composed of several thousands up to billions of gates and interconnections, performance posed an urgent issue from the very beginning of the development of HAL.

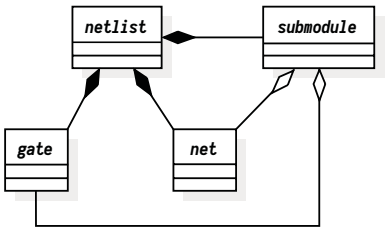


Figure 2: Simplified HAL netlist library class diagram

Therefore, the low-level programming language C++17 was chosen to implement the underlying core system. Here the netlist library, which represents the data structure for all elements of a netlist, constitutes the crucial component. The class diagram in Figure 2 depicts the relationship between the core classes.

In contrast to off-the-shelf graph libraries, the netlist library has the following distinct properties, which are specifically designed for netlist processing.

Gate Each gate object has a gate type (e.g., NAND, NOR, ...) dynamically assigned based on the underlying gate library while parsing the netlist. Additional information, e.g., LUT configuration strings, FF init values, etc., are stored directly in the gate object.

Net In contrast to classical edges with a single source and sink a net in our library allows to have multiple sinks.

Submodule To add hierarchy information during the reverse engineering process additional submodules can be defined. Each submodule lists the gates and nets belonging to the submodule.

In addition to the core module, the netlist library introduces an event system allowing other components to be notified when the underlying data model changes its information. This is specifically necessary for interactive components like the Graphical User Interface (GUI).

The Plugin System. Due to the collision between the requirement of working on netlist reversing projects under an Non Disclosure Agreement (NDA) and the goal to provide a collaborative open-source framework to the community, we decided to introduce a plugin system. It allows leaving the core parts of the system public while project-specific elements can be placed in a plugin without the necessity to publish them. The plugin system is realized through C++ dynamic libraries which are loaded on demand by the core. This allows for straightforward parallelization of computation-intense tasks, for example via OpenMP.

As an example, we provide a plugin for dynamic graph analyses called `graph-algorithm` which allows further processing of a netlist in HAL using the Boost Graph library[2].

[2]Boost Graph Library, https://www.boost.org/doc/libs/1_66_0/libs/graph/doc/index.html

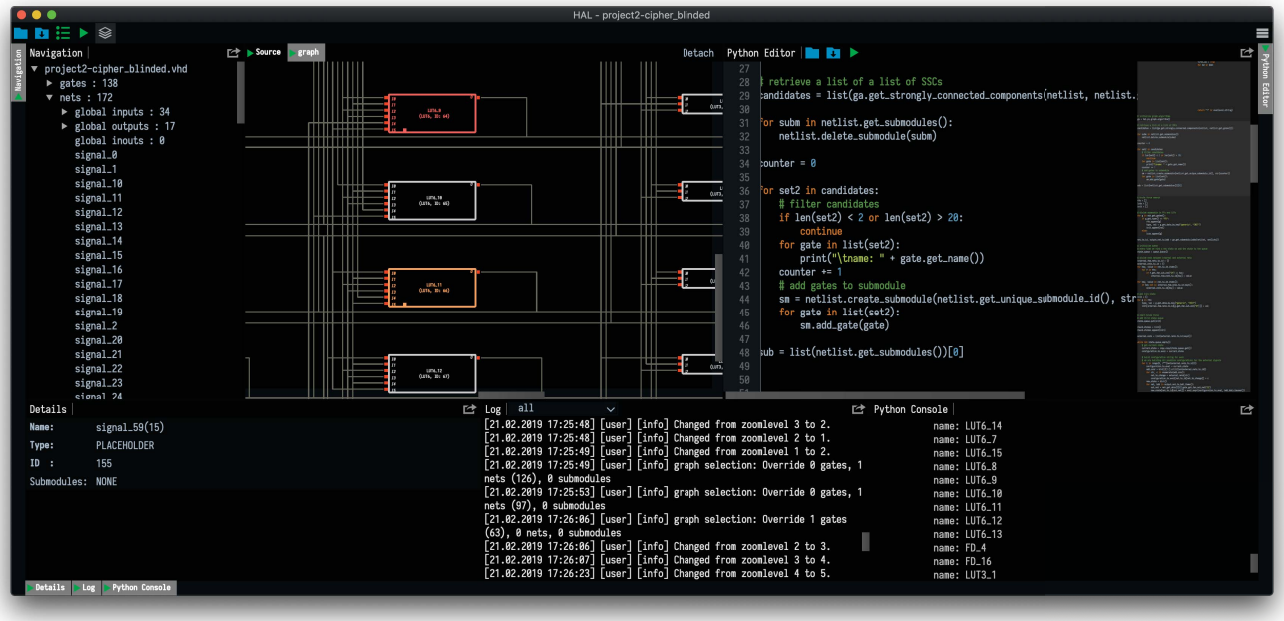


Figure 3: Screenshot of the HAL Graphical User Interface during the Reverse Engineering Process

The Graphical User Interface. While algorithmic analyses of netlists are a powerful tool, there are cases where the visual inspection of a design is necessary. Therefore, our Qt5[3]-based GUI provides a performant graphical representation even for large netlists and enables interactive navigation with mouse and keyboard through the graph. At the same time, a navigation pane ensures that the user always maintains an overview, while additional information about the selected netlist component is displayed in a detail pane.

A further GUI-feature is the color-based submodule highlighting, which supports users in the process of understanding the inner workings of a design in combination with self-developed plugins for algorithmic analyses.

Altogether, the GUI facilitates the reverse engineers' task to process the given information and to make sense of a formerly unknown design [30].

Python Integration. To lower the barrier of entry for new HAL users and developers, we embedded a Python shell into the GUI. The Python shell provides an efficient and intuitive approach to interact with a netlist; whereas the development of custom C++ plugins offers full flexibility, but requires more experience.

From a technical perspective, we employed pybind11[4]to map the C++ API to the smaller and simpler Python API. All function calls from the fully-featured Python 3.7.2 interpreter to the core are handled by the C++ back-end to preserve its performance advantages.

[3]Qt5, https://www.qt.io/
[4]Pybind11, https://github.com/pybind/pybind11

4 CASE STUDIES

In the following we present two case studies demonstrating the capabilities and flexibility of HAL in reverse engineering gate-level netlists.

4.1 Reverse Engineering Finite States Machines

Since an FSM controls almost every hardware design it presents a promising attack target for reverse engineers. Fyrbiak et al. [10] proposed a method for finding FSM circuits in a netlist as well as a way of retrieving the corresponding state-graph. For extracting FSMs from a netlist a plugin in HAL has been created.

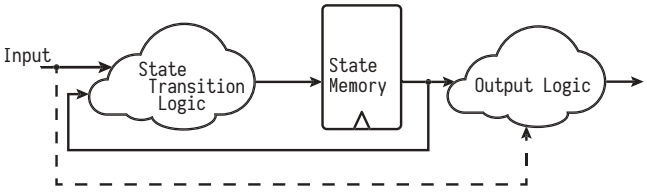


Figure 4: Block diagram of a hardware FSM (dashed line in the case of a Mealy machine) from [9]

From a mathematical perspective FSMs are equivalent to a Strongly Connected Component (SSC) (see Figure 4). There are several algorithms from graph theory that can be used to identify SSCs. In HAL we implemented Tarjan's Algorithm [26] to identify SSCs in our own `graph_algorithm` plugin. Once the plugin has been loaded, the corresponding functions can operate on the netlist and report back the

results to the main program, where they can be used for further analysis.

After the *correct* FSM circuit has been identified, the state graph can be retrieved by analyzing the Boolean logic of the state transition logic. We conducted our analysis for FPGA netlists, which incorporate mostly LUTs to realize the logic. Using the functionalities provided by `gate-decorators`, HAL offers the possibility to generate a Binary Decision Diagram (BDD) representing the logic expression for one or multiple LUT-gates. A `gate-decorator` extends the already available functions for gates provided by the `gate` class. `Gate-decorators` are specific for every gate-library. This means, for porting said method to extract FSMs from ASIC netlists, one has to provide the specific definitions of the gate-library in order to generate the corresponding BDD. With the help of the generated BDDs, we can brute-force all reachable states for reasonably bounded FSMs without the need for further libraries. Of course, computation time grows quickly with the complexity of the feedback logic and the FSM state register's size. In the end, we output the state graph as a GRAPHVIZ `.dot` file.

Circumventing FSM Obfuscation. Obfuscation describes the transformation, which obstructs high-level information without changing functionality while increasing the complexity of the reverse engineering process in mind.

Fyrbiak et al. [10] also described means of attacking several well-known obfuscation schemes operating on the FSM-level [1, 4, 6]. Since HAL offers netlist manipulation techniques, we can efficiently implement a plugin to circumvent, remove, or disable obfuscation techniques.

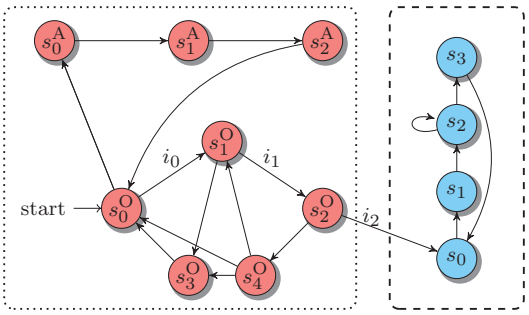


Figure 5: Obfuscated FSM using HARPOON [4, 10]

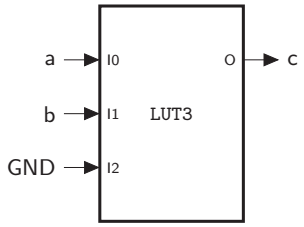
One of the most popular obfuscation schemes is HARPOON [4]. The basic idea of HARPOON - see Figure 5 - envisages a designer inserting a second FSM (highlighted in red) to protect the original FSM (highlighted in blue). The inserted FSM part has to be traversed in a certain way, using a specific input sequence called the enabling key. Every other input sequence than the enabling key will not lead to the original FSM, thus rendering the hardware design unusable for unauthorized parties [31].

Fyrbiak et al. [10] proposed a general attack idea to (i) find a HARPOON key and (ii) remove the HARPOON key from the netlist. We introduce and use these attack ideas to present various features of HAL, e.g., the netlist manipulation, and plugin features. First, with the brute-force attack described in Section 4.1, we executed the FSM detection plugin and read the HARPOON enabling key from the extracted state graph.

Second, we completely changed the behavior of the state machine and generate a manipulated netlist. For that, we use the netlist manipulation feature of HAL. Changing the initial value of the Flip-Flops from the state memory to the values of the initial state of the original FSM results in omitting the obfuscated part. This removing of the obfuscation FSMs is possible as we know the initial states from the first attack step. The manipulated netlist can be written to either Verilog or Very High Speed Integrated Circuit Hardware Description Language (VHDL). In case of an FPGAs netlist, a new bitstream can be generated with the corresponding vendor tools. HAL even allows manipulating the transition logic and thus manipulating the behavior of the state machine.

4.2 Finding Watermarks

In the context of hardware design, a watermark is a secret or hidden *message* inside a circuit that enables the owner of the design to identify his work. It is usually used in the context of IP-infringement to identify intellectual property.



I2	I1	I0	O
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	Y
1	0	1	Y
1	1	0	Y
1	1	1	Y

Figure 6: Overview of the watermarking scheme

Wallat et al. [28] proposed ways for identifying and removing a watermark scheme presented by Schmid et al. [23] for FPGAs.

The scheme makes use of the fact that sometimes not all inputs of a LUT are in use. If a LUT has an unused input, it is usually being connected to either `GND` or `VCC` – respectively logic '0' or '1' . If a LUT has an input connected to `GND` or `VCC` it results in unreachable entries in the truth table – see Figure 6 – the entries marked with `Y` are the entries that cannot be reached. Their watermarking scheme inserts a unique sequence into these unreachable entries to uniquely mark a design.

We used HAL to identify LUTs that were used for the watermarking by analyzing the LUT content of all LUTs that have `GND` or `VCC` connected. For each of these LUTs the LUT content is analyzed for entries that where not set to '0', when they cannot occur. This way the watermarking can easily be

identified. Furthermore we removed the watermarking using HAL, by manipulating the LUT's content using the netlist manipulation feature.

5 CONCLUSION

Hardware reverse engineering as the tool-of-choice to examine hardware designs for their functionality and potential manipulations, or to detect product counterfeits. Due to the lack of publicly available and fully-customizable frameworks assisting the gate-level netlist reversing process we present our gate-level netlist reverse engineering framework HAL. Furthermore we present its rich feature set providing visual and algorithmic access to gate level netlists, as well as its technical foundations to get potential users started. In an effort to involve the open-source community into the development, we release the HAL source code on https://github.com/emsec/hal under the MIT open-source license.

A main feature of HAL is the representation of the netlist as a graph which enables further graph-based analyses. In two case studies we demonstrated the manifold capabilities of HAL: First, we illustrate the creation of plugins to simplify the netlist reverse engineering process in a practical context. Second, we demonstrated how the `graph_algorithm` plugin can be applied to identify structures and modules within the flat netlist. In the end, the powerful manipulation feature shows how the behavior of a netlist can be changed to circumvent real-world obfuscation techniques.

ACKNOWLEDGMENT

The research was supported in part by ERC Advanced Grant 695022 and NSF award NS-1563829.

REFERENCES

[1] Yousra Alkabani et al. 2007. Active Hardware Metering for Intellectual Property Protection and Security. In USENIX Security Symposium.
[2] F. Benz et al. 2012. BIL: A Tool-Chain for Bitstream Reverse-Engineering. In IEEE FPL. 735–738.
[3] Swarup Bhunia et al. 2014. Hardware Trojan Attacks: Threat Analysis and Countermeasures. Proceedings of the IEEE 102, 8 (2014), 1229–1247.
[4] Rajat Subhra Chakraborty et al. 2009. HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection. IEEE Trans. CAD of Integrated Circuits and Systems 28, 10 (2009), 1493–1502.
[5] Z. Ding et al. 2013. Deriving an NCD File From an FPGA Bitstream: Methodology, Architecture and Evaluation. Microprocessors and Microsystems - Embedded Hardware Design 37, 3 (2013), 299–312.
[6] J. Dofe et al. 2018. Novel Dynamic State-Deflection Method for Gate-Level Design Obfuscation. IEEE Trans. on CAD of Integrated Circuits and Systems 37, 2 (2018), 273–285.
[7] Maik Ender et al. 2019. Insights into the mind of a trojan designer: the challenge to integrate a trojan into the bitstream. In ASPDAC 2019, Tokyo, Japan, January 21-24, 2019. 112–119. https://doi.org/10.1145/3287624.3288742
[8] M. Fyrbiak et al. 2017. Hardware Reverse Engineering: Overview and Open Challenges. In 2017 IEEE 2nd International Verification and Security Workshop (IVSW).
[9] Marc Fyrbiak et al. 2018. HAL-The Missing Piece of the Puzzle for Hardware Reverse Engineering, Trojan Detection and Insertion. IEEE Transactions on Dependable and Secure Computing (2018).
[10] Marc Fyrbiak et al. 2018. On the Difficulty of FSM-based Hardware Obfuscation. IACR Transactions on Cryptographic Hardware and Embedded Systems 2018, 3 (Aug. 2018), 293–330. https://doi.org/10.13154/tches.v2018.i3.293-330
[11] Steve Guccione et al. 2011. JBits: Java based interface for reconfigurable computing. In CCS 2011. ACM.
[12] Bernhard Lippmann et al. 2019. Integrated flow for reverse engineering of nanoscale technologies. In ASPDAC 2019, Tokyo, Japan, January 21-24, 2019. 82–89. https://doi.org/10.1145/3287624.3288738
[13] T. Meade et al. 2016. Gate-Level Netlist Reverse Engineering Tool Set for Functionality Recovery and Malicious Logic Detection. International Symposium for Testing and Failure Analysis (ISTFA) (2016).
[14] Amir Moradi et al. 2011. On the Vulnerability of FPGA Bitstream Encryption Against Power Analysis Attacks: Extracting Keys From Xilinx Virtex-Ii FPGAs. In ACM CCS. 111–124.
[15] Amir Moradi et al. 2012. Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures. In Topics in Cryptology – CT-RSA 2012. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–18.
[16] Amir Moradi et al. 2013. Side-channel Attacks on the Bitstream Encryption Mechanism of Altera Stratix II: Facilitating Black-box Analysis Using Software Reverse-engineering (FPGA '13). ACM, New York, NY, USA, 91–100. https://doi.org/10.1145/2435264.2435282
[17] Amir Moradi et al. 2016. Improved Side-Channel Analysis Attacks on Xilinx Bitstream Encryption of 5, 6, and 7 Series. In Constructive Side-Channel Analysis and Secure Design. Springer International Publishing, Cham, 71–87.
[18] Jean-Francois Nguyen. 2016. Analysing the Bitstream of Altera's MAX-V CPLDs. https://lse.epita.fr/lse-summer-week-2016/slides/lse-summer-week-2016-07-maxv_cpld.pdf
[19] Jean-Baptiste Note. 2008. debit. https://github.com/djn3m0/debit/tree/master/altera
[20] Jean-Baptiste Note et al. 2008. From the Bitstream to the Netlist. In ACM FPGA. 264–264.
[21] Khoa Dang Pham et al. 2017. BITMAN: A Tool and API for FPGA Bitstream Manipulations. In DATE. 894–897.
[22] Shahed E Quadir et al. 2016. A Survey on Chip to System Reverse Engineering. JETC 13, 1 (2016), 1–34.
[23] Moritz Schmid et al. 2008. Netlist-Level IP Protection by Watermarking for LUT-based FPGAs. In ICECE Technology, 2008. FPT 2008. International Conference on. 209–216.
[24] Pawel Swierczynski et al. 2014. Physical Security Evaluation of the Bitstream Encryption Mechanism of Altera Stratix II and Stratix III FPGAs. ACM Trans. Reconfigurable Technol. Syst. 7, 4, Article 34 (Dec. 2014), 23 pages. https://doi.org/10.1145/2629462
[25] SymbiFlow. 2017. Project X-Ray. https://github.com/SymbiFlow/prjxray
[26] Robert Tarjan. 1971. Depth-first search and linear graph algorithms. In 12th Annual Symposium on Switching and Automata Theory (swat 1971). IEEE, East Lansing, MI, USA, 114–121. https://doi.org/10.1109/SWAT.1971.10
[27] R. Torrance. 2009. The State-Of-The-Art in IC Reverse Engineering. In CHES. Springer, 363–381.
[28] Sebastian Wallat et al. 2017. A Look at the Dark Side of Hardware Reverse Engineering – A Case Study. In 2017 IEEE 2nd International Verification and Security Workshop (IVSW).
[29] Weste, Neil and others. 2010. CMOS VLSI Design: A Circuits and Systems Perspective (4th ed.). Addison-Wesley Publishing Company, USA.
[30] Carina Wiesen et al. 2018. Teaching Hardware Reverse Engineering: Educational Guidelines and Practical Insights. IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE) (2018).
[31] Carina Wiesen et al. 2019. Towards Cognitive Obfuscation: Impeding Hardware Reverse Engineering Based on Psychological Insights. In ASPDAC 2019, Tokyo, Japan, January 21-24, 2019. ACM, New York, NY, USA, 104–111. https://doi.org/10.1145/3287624.3288741
[32] D. Ziener et al. 2006. Identifying FPGA IP-Cores Based on Lookup Table Content Analysis. In Field Programmable Logic and Applications, 2006. 1–6. https://doi.org/10.1109/FPL.2006.311255