HAL—The Missing Piece of the Puzzle for Hardware Reverse Engineering, Trojan Detection and Insertion

Marc Fyrbiak, Sebastian Wallat, Pawel Swierczynski, Max Hoffmann, Sebastian Hoppach, Matthias Wilhelm, Tobias Weidlich, Russell Tessier, and Christof Paar *Fellow, IEEE*

Abstract—Hardware manipulations pose a serious threat to numerous systems, ranging from a myriad of smart-X devices to military systems. In many attack scenarios an adversary merely has access to the low-level, potentially obfuscated gate-level netlist. In general, the attacker possesses minimal information and faces the costly and time-consuming task of reverse engineering the design to identify security-critical circuitry, followed by the insertion of a meaningful hardware Trojan. These challenges have been considered only in passing by the research community. The contribution of this work is threefold: First, we present HAL, a comprehensive reverse engineering and manipulation framework for gate-level netlists. HAL allows automating defensive design analysis (e.g., including arbitrary Trojan detection algorithms with minimal effort) as well as offensive reverse engineering and targeted logic insertion. Second, we present a novel static analysis Trojan detection technique ANGEL which considerably reduces the false-positive detection rate of the detection technique FANCI. Furthermore, we demonstrate that ANGEL is capable of automatically detecting Trojans obfuscated with DeTrust. Third, we demonstrate how a malicious party can semi-automatically inject hardware Trojans into third-party designs. We present reverse engineering algorithms to disarm and trick cryptographic self-tests, and subtly leak cryptographic keys without any a priori knowledge of the design's internal workings.

Index Terms—Hardware Reverse Engineering, Hardware Trojans, Hardware Trojan Detection

1 Introduction

Hardware is the root of trust in virtually any modern computing system, ranging from traditional computer networks using the Internet of Things (IoT) to military systems. Malicious manipulations of the underlying hardware can have catastrophic consequences for the safety and security of the target systems. However, modern Application Specific Integrated Circuit (ASIC) design and fabrication processes are heavily globalized with various (untrusted) stakeholders, including Intellectual Property (IP) providers and off-shore foundries [1]. Both are able to inject malicious circuitry prior to fabrication with devastating consequences to system security, either for their own advantage or as a requirement by nation-state adversaries. For market-dominating Static Random Access Memory (SRAM)-based Field Programmable Gate Arrays (FPGAs), the situation is similarly gloomy, since protective bitstream encryption can be invalidated for the majority of currently employed families [2], [3], allowing adversaries to inject malicious circuitry [4]. We also note that most current low-cost FPGA families do not offer bitstream encryption.

Despite intense work on hardware Trojans in the scientific community, see Bhunia *et al.* [5] for a comprehensive overview, there has only been a scant treatment regarding the practicability of actually inserting hardware Trojans into designs, since the vast majority of work supposes access to the Hardware Description Language (HDL) source code or neglects the crucial step of reverse engineering, see [6], [7], [8], [9], [10], [11], [12].

Even though reverse engineering has attracted little scrutiny from the scientific community, understanding its considerable complexity is indeed essential. This holds for developing sound threat assessments of hardware Trojans, as well as providing guidelines for developing countermeasures such as hardware obfuscation and physical design obfuscation [13], [14]. Also, understanding hardware reverse engineering is crucial for developing countermeasures against the pressing problem of IP theft. Similar to the hardware design process, hardware reverse engineering requires frameworks and tools to automate custom time-consuming tasks and simplify the steps for a human analyst. Given that nation-state adversaries, who arguably pose the most credible threat regarding hardware Trojans, might have developed hardware reverse engineering frameworks and tools, a better understanding of such frameworks is crucial for the security community at large. While hardware reverse engineering frameworks and tools exist in the industrial sector [15], [16], to the best of our knowledge, no publicly available reverse engineering and manipulation framework for gate-level netlists exists. We would like to note that several academic reverse engineering tools exist [17], [18], [19], however, these are not frameworks that are gate-level library agnostic (e.g., usable for both FPGAs and ASICs), they do not provide built-in extensibility to interactively integrate custom tools, nor do they have a rich-featured interactive

- *M. Fyrbiak, P. Swierczynski, M. Hoffmann, S. Hoppach, M. Wilhelm, T. Weidlich, and C. Paar were with the Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany. E-mail: {prename.surname}@rub.de*
- *S. Wallat, R. Tessier, and C. Paar were with the University of Massachusetts Amherst, USA. E-mail: {swallat, tessier}@umass.edu*

Manuscript received July 31, 2017;

Graphical User Interface (GUI), suited for manual analysis. These are relevant requirements for reverse engineering.

Goals and Contributions. In this paper, we focus on reverse engineering of high-level information from low-level unfolded, placed-and-routed, gate-level netlists. Our goal is to demonstrate actual netlist reverse engineering and manipulation capabilities for offensive and defensive applications under realistic assumptions. To this end, we first address the lack of netlist-level reverse engineering frameworks in the open literature and present HAL, a holistic framework to support and automate custom time-consuming tasks such as Trojan insertion and detection, or assessment of IP violations. HAL's primary purpose is to facilitate hardware security research, allowing researchers to focus on the innovative aspects of their work and unify collectively-acquired knowledge. Moreover, we present our novel static analysis hardware Trojan detection technique, ANGEL, which outperforms a previously-proposed static analysis detection method, FANCI [20]. In particular, we demonstrate that we are able to automatically find Trojan triggers which were obfuscated with the Trojan obfuscation scheme DeTrust presented at CCS'14 [10], as well as k-XOR-LFSR Trojans [21]. We then present multiple reverse engineering and manipulation case studies with a focus on how gate-level netlists of cryptographic designs can be surreptitiously weakened. In particular, we demonstrate how security-relevant parts can be semi-automatically reverse engineered, and how custom-tailored malicious circuitry can be injected to invalidate the system security. These manipulations are possible even after validation processes such as formal verification, code reviews, or functional tests have been performed. Our offensive case studies focus on representative real-world cases and are supported by an evaluation with multiple FPGA families and a variety of synthesis optimization strategies. In summary, our main contributions are:

- **Netlist Reverse Engineering Framework.** We present the design and implementation of HAL, an interactive framework for virtually any user-defined gate-level library, including ASIC and FPGA libraries. HAL supports and automates reverse engineering tasks and enables tailored manipulations with ease. HAL also assists users in debugging, testing, and structural analyses to make sense of large and complex gate-level netlists. A core feature of HAL is its extendability, and we demonstrate that the development of custom tools for reverse engineering, Trojan detection and Trojan injection is surprisingly fast and efficient.
- **Hardware Trojan Detection.** We present our novel hardware Trojan detection technique ANGEL which is based on Boolean function analysis and graph neighborhood analysis. In particular, we demonstrate that graph neighborhood analysis considerably reduces the crucial false-positive rate by several factors and can detect Trojans armed with the obfuscation scheme DeTrust as well as k-XOR-LFSR Trojans.
- **Low-level Hardware Trojan Insertion.** We detail the many-faceted workflow of semi-automated hardware Trojan insertion with accompanied reverse engineering under realistic assumptions. In several case studies, we demonstrate how meaningful Trojans can be injected into third-party gate-level netlists. Our custom-tailored hardware Trojans semi-automatically invalidate security measures of cryptographic implementations, including bypassing self-tests and leaking crypto keys.
- **Novel Reverse Engineering Techniques.** Our case studies include novel reverse engineering techniques to algorithmically disclose security-relevant parts such as cryptographic self-tests and interfaces of cryptographic implementations. We provide results for numerous cryptographic implementations, a variety of FPGA families, and several design optimization goals.

We believe that the insights provided by HAL are in particular relevant since it can be speculated that large-scale adversaries, such as nation-states will most likely invest in similar tools for hardware manipulations, and the work at hand provides a guideline for threat assessment and countermeasure design.

2 Background and Related Work

Since HAL processes gate-level netlists for reverse engineering, some fundamental background on hardware security and reverse engineering is required to understand the mechanics of HAL. To this end, we first discuss the threat model, i.e. how an adversary obtains gate-level netlists in several real-world scenarios.

2.1 Threat Model

We assume an adversary with access to the flattened (placed and routed) gate-level netlist without any a priori knowledge of the design's internal workings. More precisely, the adversary has no information of module hierarchies, synthesis options, or names of gates and signals. The high-level goal of the adversary is to inject a hardware Trojan into the gate-level netlist. To this end, the adversary has to reverse engineer (parts of) the design in order to identify the relevant gates and signals where the hardware Trojan has to be attached. The gate-level netlist can be obtained through several means: (1) chip-level or layout reverse engineering [22], [14] in the case of ASICs, (2) bitstream-level reverse engineering [4], [23] in the case of FPGAs, or (3) directly from the IP provider [1].

Note that our threat model is consistent with prior research on hardware security [24], [25], [1], [5], [26].

2.2 Gate-Level Netlist Reverse Engineering

Modern digital circuit design is typically realized at the Register Transfer Level (RTL) which models the signal flow among registers. Logic synthesis tools convert RTL descriptions to gate-level netlists, i.e. a list of gates and their interconnection. Subsequently, place and route algorithms process the gate-level netlist and determine where gates are placed and how the interconnections are routed. From a reverse engineering perspective, valuable information is lost during this translation: module boundary information and hierarchy information [17], [27]. In addition, diverse optimizations are performed to achieve a predefined optimization goal such as improved area or timing. Thus, a

reverse engineer targets this information to then recover crucial high-level information about the design.

In 1999, Hansen *et al.* [28] described several steps for a human reverse engineer to retrieve high-level information from gate-level netlists such as the identification of common library structures and the detection of recurrent modules. Shi *et al.* [29] reported a technique to algorithmically extract Finite State Machines (FSMs). Later, Shi *et al.* [30] described a method to extract diverse functional modules from a gate-level netlist. In 2012, Li *et al.* [31] developed a technique to match unknown sub-circuits against library components based on pattern mining of simulation traces and model checking. In further work, Li *et al.* [32] described how word-level structures can be uncovered automatically. Based on this work, Subramanyan *et al.* [33], [17] extended the arsenal of reverse engineering algorithms by extraction of functional components such as register files and adder units. Functional identification also requires identifying the correct order of input bits (of the component under inspection), i.e. least to most significant bit. Thus, a reverse engineer has to brute-force the correct permutation or derive it from already reversed components. Gascón *et al.* [34] addressed this problem with a template-based solution. Wallat *et al.* [35] presented several insights on offensive aspects of reverse engineering such as the removal of watermarks for IP protection and the manipulation of stream ciphers.

While previous works focused primarily on ASICs, we additionally address FPGA platforms in our case studies. Furthermore, we take a step forward and demonstrate how reverse engineering and manipulation techniques act jointly to inject hardware Trojans in security-critical circuitry of third-party gate-level netlists.

2.3 Hardware Trojans

Since an initial report by the US DoD in 2005 [36], the scientific community has extensively researched offensive and defensive aspects of malicious hardware manipulations, see Bhunia *et al.* [5] for a comprehensive overview. Typically, a hardware Trojan consists of a *payload* delivering the malicious functionality (e.g., leakage of cryptographic keys or denial of service) and an optional payload activating *trigger* (e.g., a counter or sensor). Note that if a trigger is not used, the Trojan is always active.

Defensive research focuses on the detection of hardware Trojans based on diverse characteristics such as physical attributes, trigger features, and payload features [37]. In order to detect characteristics, various approaches based on side-channel analysis [5], and static or dynamic design analysis [38], [20], [39], [40], [21], [41] have been proposed, see Section 4.3 for more details on the different detection techniques. Several works targeted manipulations at layout-level design methodologies such as dopant-level Trojans [9], analog Trojans [12], or parametric Trojans [11]. In contrast to the vast amount of defensive research, considerably less public research focuses on offensive aspects. The majority supposes access to the HDL source code or neglect reverse engineering of crucial high-level information (see [6], [7], [8], [10]). Our work addresses this important gap and we demonstrate how hardware Trojans can be injected at the gate-level even after validation processes have been performed.

2.4 FPGA Security

In order to analyze and potentially manipulate an FPGA design post-deployment, an adversary has to analyze the bitstream configuring the FPGA. In the case of market dominating SRAM-based FPGAs, the bitstream is stored in an external non-volatile memory (e.g., a flash chip). Thus, the bitstream can be extracted with relatively low effort by directly dumping the memory contents or via wiretapping during boot-up.

To counteract IP theft, various FPGA manufacturers have included bitstream encryption in their product series. Unfortunately, past research has demonstrated that the majority of these schemes are vulnerable to Side-Channel Analysis (SCA) attacks, i.e. that the secret key can be extracted [2], [3], [42]. This enables an adversary to obtain a decrypted bitstream which can be manipulated and re-encrypted. Even though the bitstream is a proprietary encoding of a gate-level netlist, it has been shown that this encoding can be (automatically) reverse engineered [43], [44], [45], [46], [23].

Despite intense research in FPGA security, only a few works have focused on bitstream-level manipulations to practically extract cryptographic keys from block ciphers [47], [48], [4], [49]. More specifically, these works addressed the analysis and exploitation of Block-Ram and Look-up table (LUT) configuration. In contrast to the limited bitstream-level manipulation attacks, we focus on an adversary with access to a gate-level flattened netlist, i.e. the entire hardware configuration.

3 HAL - Design and Implementation

We now describe HAL's overall architecture, workflow, and implementation. To this end, we want to stress that HAL itself is not a tool but a comprehensive framework that can be used to create tools, a common task during reverse engineering.

3.1 HAL System Architecture

HAL was written according to modern software design and architecture standards to achieve easy maintainability, extendability, and high modularity. Therefore, HAL consists of several separated building blocks, each focusing on a logical feature set. In the following we outline the workflow of HAL, before providing more detail on the main building blocks. To guide the reader, we will refer to the numbered circles of Figure 1 which provides an overview of the workflow of HAL.

HAL - General Workflow. The user invokes HAL with a gate-level netlist ❶. HAL uses one of its parsers ❷ (e.g., VHDL or Verilog) to transform the input netlist into its internal graph representation ❸. After this translation step, user-defined plugins ❺ can be invoked via the plugin manager ❹ to automatically analyze and possibly manipulate the gate-level netlist. All changes to the graph throughout the plugin operations including meta data added by plugins or the user (e.g., meaningful names and hierarchy information) are synchronized with a local database ❿. To further support the user, the whole workflow is also accessible via an interactive GUI ❾ and an interactive Python shell ❿. When all requested plugins and tasks have been processed, the graph may be written back ❻ to a

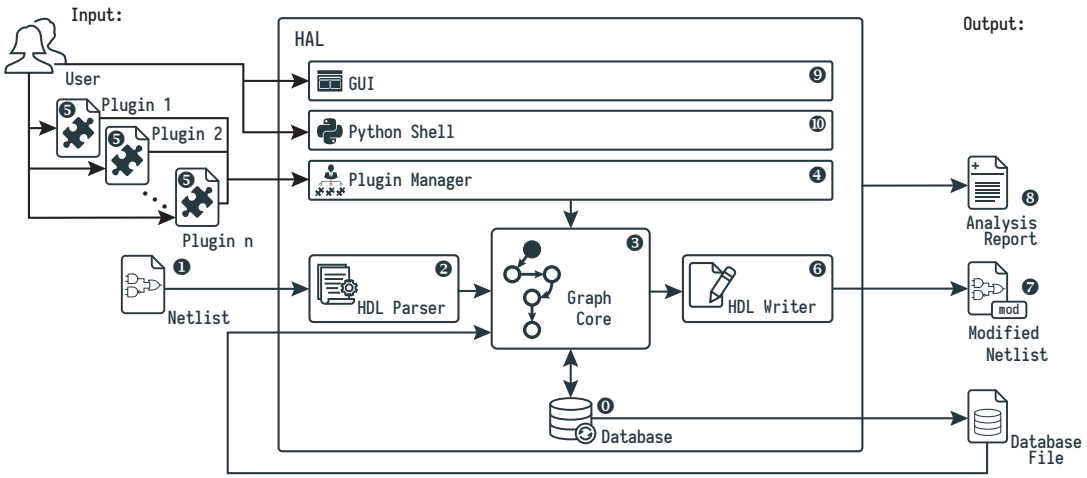


Figure 1. Overview on HAL's architecture and workflow.

gate-level netlist for synthesis or simulation ❼ in any of the supported HDL languages.

3.2 HAL- Building Blocks

HDL Parser and Writer. HAL transforms an input netlist ❶ (e.g., in VHDL or Verilog) into a directed multi-graph representation of the design. To be more precise, a gate-level netlist includes a series of gates (nodes) and how they are connected via nets (directed edges) where a gate A may have two output ports and both connect to the input of gate B (multi-digraph). Typically, a netlist includes a set of *atomic* gates defined in a gate-level library which specifies their behavior. However, in cases of chip-level reverse engineering the gate-level library is often not known beforehand and is disclosed during the process itself. To support incorporation of custom netlist libraries and even HDL languages, we developed an extensible parser and writer interface which is independent of the gate-library and source language. Currently, we support all Xilinx FPGA and ASIC gate libraries for the TrustHub benchmark suite [50]. Adding libraries is straightforward and can be done without recompiling HAL.

After the analysis and manipulation step, a reverse engineer may be interested in a synthesizable netlist containing the modified graph ❼. To provide this functionality, HAL includes multiple HDL file writers ❻, including VHDL and Verilog, which transform the modified design back to its netlist representation in the user-preferred supported HDL language. For example, if the input is a VHDL netlist we can also output a Verilog version which is handy since several open-source tools such as Verilator [51] do not support VHDL.

Graph Core. The operational heart of HAL is the graph core ❸ which allows netlist exploration. Our graph representation is a high-level abstraction of any arbitrary gate-level netlist independent of the underlying gate library or HDL language, analogous to intermediate representations used in software program analysis. Note that such a step is favorable in practice since automated techniques can be developed on the high-level representation rather than a single technique for each gate library. Furthermore, dedicated algorithms from graph theory can be employed, leveraging existing research from math theory. In essence, the graph core provides graph traversal functionalities and methods to edit the graph itself. The graph can be reshaped by adding new gates and nets or removing existing ones. Moreover, the grouping and annotation of gates and nets into subgraphs (modules in HDL) is supported which is important for reconstruction of a design's module hierarchy.

Furthermore, we implemented a dynamic and generic decorator interface to add user-defined behavior and state to gates and submodules during HAL execution. For example, we developed decorators to provide Binary Decision Diagram (BDD) representations of gates and special memory access functionality for LUT gates. Since we are using C++, static inheritance (instead of decorators) is not a favorable solution to dynamically attach additional functionality.

On top of these fundamental operations, the Application Programming Interface (API) provides the user with access to high-level graph analysis algorithms (e.g., Dijkstra's Shortest Path, Strongly Connected Components). Since all plugins typically make heavy use of the graph core, it is optimized for speed and failure safety.

To enable persistent storage of revealed information and to facilitate joint reverse engineering on a specific design, we integrated a database synchronization engine ❿. The database not only stores the graph itself, but also any user-defined meta data such as names and hierarchy information. Since textual representations of flattened gate-level netlists are in the range of several megabytes, we decided to employ a performance-optimized NoSQL database and implemented a custom key format to store arbitrary user data. With our database engine we enable collaborative analysis of the same netlist and snapshot generation since the database files can easily be shared among a team of users or saved as a backup.

Plugin System. To separate the development of the core framework from user-defined applications and algorithms, we use a plugin-based system architecture to dynamically include external code. Here, a plugin-based approach is favorable since this approach is highly extensible and HAL itself does not have to be recompiled when new plugins are implemented. We use a plugin manager ❹ which handles registration, loading, usage, and eventually unloading of

plugins during run-time. Multiple plugins can be executed in parallel or consecutively invoked to automatically engage each other.

Analysis Report. Using a sophisticated logging system the user can create rich analysis report files ❽ for result logging or as debugging information. The logging system was created with support for multiple channels and severity levels, as well as intuitive output formatting.

GUI. An engineer can already accomplish a lot by using HAL only via its command line interface with the plugin system. However, with textual output alone it can be difficult to make sense of huge designs with potentially billions of gates. To mitigate the manipulation of overwhelming amounts of textual information, we included an interactive GUI to enable visual representation of the gates and nets of the processed graph ❾. For optimal presentation, we created a unique graph view layout tool from the ground up to meet our high requirements of interactive design exploration. Note that layout planning of large gate-level netlist graphs is a challenging problem since the vertex and edge arrangement has to be computed quickly and focused on comprehensibility (e.g., to support the mental map for manual reverse engineering). Therefore, we integrated a generic interface to support multiple graph layout algorithms as different layout techniques focus on different visual representations. For example, we integrated an orthogonal layouter, which arranges a graph in a rectangular 2D grid, as well as a hierarchical layouter that leverages additional information about the distance of nodes to I/O ports.

The GUI is also enhanced with interaction. The graph can be visually traversed by moving from pin to pin by clicking on the graphical representation or using keyboard shortcuts. Additionally multiple docks provide rich information about selected components and plugin-generated annotations. For example, a reverse engineering plugin that automatically detects a cryptographic module can adjust the color of identified gates and nets to aid the analyst. We stress that the GUI is of immense interest for a human reverse engineer, since making sense of a complex design is notably easier with a visual representation than just with textual information.

Python Shell. To allow simple execution of arbitrary core functionalities, for example for testing or even batch execution of long running analysis tasks, we integrated an interactive Python3-based shell console widget ❿ into HAL. Our Python shell allows access to the fast C++ HAL API from within the Python interpreter by mapping each core function from C++ to Python. This enhances the *static* plugin system with interactive code execution to improve the usability during semi-automated design exploration.

Since we believe that HAL aids researchers in analyzing their designs and because of its rich set of features and the optimized core, we plan to publicly release HAL to the research community.

3.3 HAL Implementation

We implemented HAL in C++14 due to its efficiency and high performance capabilities that are especially critical for processing large hardware designs consisting of hundreds of thousands of gates. To keep HAL maintainable and extensible, we focused on clean and well documented code, as well as the predominant design principles of software architecture.

Software Libraries. We employ several components of the BOOST library (version 1.58), namely the BOOST GRAPH LIBRARY (BGL), and the Boost Filesystem. The BGL forms the backbone of our graph core as it already provides a rich set of graph algorithms. To ease functional analysis we use the BUDDY library (version 2.4) [52] to automatically generate BDD dynamic extensions for single gates or entire combinational subgraphs. For database serialization, we decided to use KYOTO CABINET, a collection of fast, serverless, NoSql database types which operates cross-platform so that database files can be easily exchanged among engineers. The GUI is built on top of the QT5 (version 5.6) application framework. Qt is also platform independent and integrates well with C++14. The interactive Python shell is built using PYTHON3 (version 3.6) and PYBIND11 (version 2.1.1) [53] to connect the C++ functions of the HAL API with Python.

To manage the build process and dependencies platform-independently, we employ the cross-platform build management tool CMAKE (version 3.6). It generates configured build files for GNU MAKE and other build systems such as NINJA. For the build process itself, we support both GCC (version 6.3) as well as LLVM (version 4). Supporting multiple compilers also results in more robust code, as the compilers perform different optimizations and provide differing output.

Currently, HAL is supported on Ubuntu, Arch Linux and macOS. Note that Microsoft Windows support is prepared, but not fully functional yet.

4 GATE-LEVEL TROJAN DETECTION - ANGEL

We now present our novel static analysis technique ANGEL (**A**nalyzing the **N**eighborhood of **G**raphs to **E**xpose **L**eakers) based on (1) Boolean function analysis, and (2) graph neighborhood analysis.

4.1 ANGEL- Technique

ANGEL builds on previous state-of-the-art research in static analysis hardware Trojan detection, namely FANCI originally presented at CCS'13 [20], [54]. Similar to FANCI we focus on detection of weakly-affecting inputs through Boolean function analysis, but additionally consider the neighborhood of combinational gates for each gate to address fundamental limitations of FANCI. To this end, we first sketch the idea of the Boolean function analysis and subsequently we present the novel idea of incorporation of the graph neighborhood. In addition, we want to emphasize that static analysis is indeed a powerful tool since it does not rely on a golden model or verification tests which is favorable in practice since fewer potential weak-points for attackers have to be trusted.

Boolean Function Analysis. To estimate the impact of an input signal on a corresponding output of a gate g, FANCI proposed a so-called *control value* CV_o which is a vector for the output o of Boolean differences BD_o for each input i standardized by the number of inputs I, i.e. $CV = \mathsf{BD}_o(i) \cdot 2^{-I}$, see [10]. The Boolean difference $\mathsf{BD}_o(i)$ computes the total number of patterns under which flipping input i results in a change of the output o. For example, let us consider a

simple AND gate with 3 input signals (i_0, i_1, i_2) and 1 output signal: the Boolean difference is $\frac{2}{2^3} = 0.25$ for each input signal, since there is a difference for two input patterns, i.e. 011 and 111 for i_0, 101 and 111 for i_1, and 110 and 111 for i_2.

Since the complexity of ANGEL (and FANCI) exponentially grows with the number of inputs for a graph cut (or a gate for FANCI), we utilize the technique developed in the original FANCI work: we approximate the control vector computation by choosing a certain number of inputs uniformly (e.g., 2^{15}) and compute the Boolean difference for these values to keep analysis time practical.

Graph Neighborhood Analysis. Since a Boolean difference computation on the single gate, as used in FANCI [20], [54], results in a high false-positive rate, as demonstrated by Zhang *et al.* [10], we consider the neighborhood of the combinational gate as well. To this end, we determine a d-feasible graph cut [17] for each gate g. In other words we apply a backward breath-first search starting from g for depth d. Afterwards, we compute the Boolean difference not on the individual gate-level, but rather on the overall neighborhood of each gate. Note that the consideration of local predecessors is favorable since a Trojan trigger is typically implemented with multiple gates. Hence, an analysis of the Boolean difference of multiple coherent gates will increase the detection probability in case of low controllability and simultaneously decrease the chance that genuine gates are flagged as malicious.

Ignoring Sequential Stages Boundaries. Due to advances in hardware Trojan design research, a Trojan trigger may be spread across multiple sequential stages [10], [21]. To cope with such obfuscation, we simply ignore the sequential registers and latches and build the graph cut using predecessors of data input ports. In this way, we are able to track the Boolean difference of local combinational gates even if they are (intentionally) in different sequential stages. Note that genuine circuits typically do not possess a low controllability across sequential stages, hence we expect that the *false-positive* rate does not increase if we ignore sequential stages.

We want to emphasize that a similar idea was noted as FANCIX by Haider *et al.* [21], but they did not perform an evaluation since they proposed "to monitor the circuits up to multiple sequential stages at a time, while ignoring any FFs in between" which consequently results in a high computational complexity. We build upon this idea and demonstrate that with an incorporation of the local neighborhood of a predefined depth d, the computational complexity is still practical on commodity hardware while simultaneously providing a low *false-positive* detection rate.

Algorithm 1 formalizes the idea of ANGEL. In line ❷ we compute a feasible graph cut of depth d while ignoring any Flip Flops (FFs) or latches in between. Lines ❸ - ❾ determine the Boolean difference for each cut. To this end determine_truth_table combines all Boolean functions of the graph cut into a single large truth table. We used the *mean-and-median* heuristic ❼, since it performs best [20].

Implementation. We now want to highlight several steps that accelerate the computation of ANGEL. First, each gate ❶ can be analyzed in parallel, as well as the computation of the Boolean difference for each input signal ❻. Hence, this approach scales with more computation power. Second, we store the heuristics for each cut so that we can perform the heuristic check ❽ for multiple threshold values, since it only involves comparison of real values (which is faster than the computation of the Boolean difference).

Algorithm 1 ANGEL

Input: D - Design gate-level netlist
Input: d - Depth
Input: t - Threshold
Output: $\mathcal{S}$ - Set of suspicious gates

```
1: for gate g ∈ D do
2:     cut c ← get_feasible_graph_cut(D, g, d)
3:     truth table t ← determine_truth_table(c)
4:     for output o of g do
5:         control vector CV ← ∅
6:         for input i of c do
7:             CV.push_back(compute_heuristic(c, i, o))
8:         if check_heueristic(CV) < t then
9:             S ← S ∪ {g}
10: return S
```

Before we present and discuss the results of our evaluation, we briefly sketch two state-of-the-art Trojan design strategies that aim for increased stealthiness by applying special constructions to the Trojan so that automatic detection algorithms are deceived.

DeTrust Hardware Trojans. Zhang *et al.* [10] presented DeTrust, a systematic way to design hardware Trojans which could not be detected by FANCI. The general idea is to hide the combinational Trojan trigger in several sequential stages. Zhang *et al.* reported that a threshold of around 0.1 exposes several Trojan related gates, however it suffers from a large amount of *false-positive* gate detections. Note that ignoring sequential stages in our graph cut determination specifically handles these DeTrust obfuscation Trojans. Analogous to Salmani [41], we realized the DeTrust obfuscation by inserting additional FFs at the output of each Trojan trigger gate.

k-XOR-LFSR Hardware Trojans. Haider *et al.* [21] presented the k-XOR-LFSR hardware Trojan design strategy to construct stealthy triggers with implicit malicious behavior. Basically, an Linear Feedback Shift Register (LFSR) consisting of k registers is leveraged to design a counter so that several *selective connections* of the LFSR state determine the trigger condition for the Trojan. Hence, the adversary is able to design more complex trigger conditions and a higher signal dimension, i.e. number of wires used to trigger the payload. For comparison to Salmani [41], we use the same 4-XOR-LFSR Trojan that leaks data for a specific LFSR state, see [41] Figure 10 (a), which was originally described by Haider *et al.* [21].

4.2 Evaluation

To demonstrate the efficiency of ANGEL, we used benchmarks from the popular TrustHub suite [50]. Table 1 shows the results of our evaluation for four TrustHub designs, (1) s15850, (2) s35932, (3) s38417, and (4) s38584. Design (1) has 1666 combinational gates and 517 sequential gates, (2) has 3717 comb. gates and 1729 seq. gates, (3) has 3739 comb.

Table 1
Evaluation of the identification accuracy of ANGEL compared to FANCI [20] for hardware Trojans equipped with DeTrust (see Section 4.2 for details). The ✓ symbol indicates that (parts of) the Trojan were identified, the ✗ symbol indicates that no part of the Trojan was identified.

Design	Defense	Threshold to Detect Trojan	Computation Time	Number of Suspicions Gates in % for Threshold t							
				t = 0.0001		t = 0.001		t = 0.01		t = 0.13	
s15850	FANCI	2^{-3}	1.81 s	0	✗	0	✗	0	✗	27.8	✓
	ANGEL ($d = 2$)	2^{-8}	13.59 s	2.4	✗	3.2	✗	8.2	✓	25.0	✓
	ANGEL ($d = 3$)	2^{-11}	4.68 m	4.4	✓	5.5	✓	9.7	✓	21.62	✓
s35932	FANCI	2^{-3}	3.37 s	0	✗	0	✗	0	✗	15.6	✓
	ANGEL ($d = 2$)	2^{-8}	10.25 s	0.01	✓	0.1	✓	0.3	✓	24.8	✓
	ANGEL ($d = 3$)	2^{-12}	18.41 s	0.1	✓	4.7	✓	4.8	✓	48.8	✓
s38417	FANCI	2^{-3}	4.34 s	0	✗	0	✗	0	✗	42.6	✓
	ANGEL ($d = 2$)	2^{-7}	52.91 s	2.7	✗	5.0	✗	13.0	✓	34.3	✓
	ANGEL ($d = 3$)	2^{-11}	1.48 h	7.4	✓	11.3	✓	16.9	✓	23.2	✓
s38584	FANCI	2^{-3}	5.34 s	0	✗	0	✗	0	✗	30.1	✓
	ANGEL ($d = 2$)	2^{-6}	44.25 s	3.2	✗	4.5	✗	10.3	✗	27.2	✓
	ANGEL ($d = 3$)	2^{-11}	10.63 m	3.6	✓	4.2	✓	6.7	✓	13.6	✓

gates and 1602 seq. gates, and (4) has 5202 comb. gates and 1282 seq. gates prior to obfuscation. Overall, we see that the *false-positive* rate decreases by several magnitudes when we increase the depth, so that the number of suspicious gates is in a range that can be analyzed manually. Our results on FANCI are similar to the evaluation results of Zhang *et al.* [10] for the threshold 0.13 (*false-positive* rate around 30% to 40%, cf. Figure 10 (a) in [10]). Note that minor deviations occur since the control value approximation is non-deterministic and we did not have access to the original implementations.

The table compares the identification accuracy of FANCI to the accuracy of ANGEL where the output of FANCI can be seen as the output of ANGEL with depth $d = 1$. It can easily be seen that ANGEL outperforms FANCI at higher depths. Additionally, our results for threshold $t = 0.01$, $t = 0.001$, and $t = 0.0001$ show that ANGEL is able to successfully detect the malicious circuity even for thresholds where FANCI is not able to detect any Trojan. We selected the exemplary threshold values 0.13 for two reasons: First, FANCI requires a threshold of > 0.125 to identify the Trojans thus we rounded this value up to report the number of *false-positive* detected gates. Second, this threshold is crucial for Trojans designed with the obfuscation DeTrust [10], which we briefly discuss in Section 4.3.

k-XOR-LFSR Hardware Trojans. We also evaluated ANGEL with respect to the 4-XOR-LFSR Trojan (described in the previous section). Note that the number of suspicious gates is similar to the DeTrust designs for FANCI and ANGEL with depths $d = 2$ and $d = 3$, since we only change the relatively small Trojan and thus we deliberately did not provide an additional table. FANCI was only able to detect the selective connections of the 4-XOR-LFSR Trojan with a threshold of 0.13, however, for this threshold the number of suspicious gates is rather high (similar as for the DeTrust Trojans), i.e. around 30% - 40%. ANGEL was able to successfully detect the Trojan for both depths $d = 2$ and $d = 3$ for all threshold values while resulting in a far lower *false-positive* rate, i.e. 4.3% for s15850 with $d = 3$. Note that an additional DeTrust obfuscation for the selective connections would not increase the stealthiness, since ANGEL simply ignores sequential stage boundaries.

As demonstrated in Table 1, ANGEL significantly reduces the *false-positive* rate and thus enables automatic detection by static analysis for DeTrust obfuscated Trojans. This observation is also true for the k-XOR-LFSR Trojan.

4.3 Discussion

DeTrust and k-XOR-LFSR Trojans. As noted before, our evaluation results regarding FANCI are the same as provided by Zhang *et al.* [10], cf. Figure 10 (a) in [10]. Since ANGEL with depth $d = 1$ basically performs the same evaluation as FANCI, both techniques yield the same number of suspicious gates in this case. Note that minor variations exist since the control vector approximation is non-deterministic and hence results differ across multiple executions. For higher depths, ANGEL significantly outperforms FANCI. In addition, we want to highlight another static analysis strategy for k-XOR-LFSR Trojans. Recently, Wallat *et al.* [35] demonstrated how LFSRs can be automatically extracted from gate-level netlists. Note that we implemented this LFSR detection using HAL, so any LFSR structure is automatically and reliably exposed in seconds. Since HAL supports development for multiple analyses and manual inspection afterwards, an analyst can examine reports from plugins to verify whether a Trojan is present or not (e.g., using both ANGEL and LFSR detection).

Threshold. A major concern for both FANCI and ANGEL is finding an appropriate threshold value. Even though it should be a small value, to the best of our knowledge, no automated means to approximate an optimal threshold exists. Our implementation aids the identification of a threshold for a given design since all control values are computed before heuristic checking.

Comparison to Other Static Schemes. We now discuss similarities and differences between ANGEL and two state-of-the-art static analysis Trojan detection techniques, namely (1) correlation-based clustering [40], and (2) COTD [41]. Similar to ANGEL, both strategies attempt to identify crucial Trojan trigger logic by computing a form of controllability, since Trojan triggers are typically associated with gates that possess low controllability. However, both techniques are orthogonal to ANGEL. For correlation-based clustering, simulation data of tests for manufacturing faults are analyzed

and a correlation-based similarity weight is determined for input/output gate values. Afterwards, a density-based clustering algorithm is used to flag outliers, i.e. potential Trojan trigger gates. As pointed out by Salmani [41], the accuracy depends on observing sufficient signal activity. For COTD, a controllability and observability value is determined by SCOAP and afterwards an unsupervised clustering analysis splits signals into malicious and genuine lists. The complexities of the different strategies are ANGEL/FANCIX: $\mathcal{O}(gm2^m)$, and COTD: $\mathcal{O}(n)$ for g = number of gates, m = (sub)circuit inputs, and n = number of wires [41]. Note that the dynamic scheme HaTCh possesses a complexity of $\mathcal{O}((2n^2)^d)$, with d = signal trigger dimension. The runtime of COTD is considerably faster than ANGEL, however, ANGEL's runtime is not impractical. Even though ANGEL possesses the exponential factor m, we leverage the approximation of the control value computation, thus this exponential factor is bound (e.g., 2^{15}) and the complexity is bounded by $\mathcal{O}(gm)$.

Lastly, we want to emphasize that with the assistance of HAL, we were able to implement a high-performance, gate-level library agnostic version of ANGEL with merely several hundred lines of C++ code. Since we also plan to publish the implementation of ANGEL in addition to HAL, other researchers can build upon our implementation and focus on novel research aspects rather than redoing implementation and experiment setup work from scratch. Furthermore, with multiple accurate and reliable hardware Trojan detection strategies such as correlation-based clustering, COTD, and ANGEL, and the (semi-)automated reverse engineering capabilities of HAL, we believe that we can significantly impede the insertion of stealthy malicious Trojans.

5 Gate-level Reverse Engineering and Manipulation - Two Case Studies

We now present two offensive case studies to demonstrate HAL's applications for gate-level Trojan injection into benign third-party gate-level netlists. In particular, we introduce generic semi-automatic reverse engineering and manipulation techniques implemented with the assistance of HAL to inject hardware Trojans into gate-level netlists of cryptographic designs. More precisely, we show how to (1) trick and disarm cryptographic power-up self-tests (Section 5.1), and (2) subtly wiretap and leak cryptographic keys via unused Input/Output (I/O) pins (Section 5.2).

Cryptographic Designs. To realize security properties such as confidentiality or integrity, it is of crucial importance that the deployed cryptographic module is not compromised. Given that most cryptographic primitives in use, such as the Suite B ciphers [55], are robust against traditional attacks, i.e. brute-force and cryptanalysis, adversaries are often forced to exploit implementation attacks to undermine the security of systems and applications. Most prominent implementation attacks are SCA and Fault Injection (FI) which have been investigated in great detail in the context of hardware security in the scientific and industrial communities, see [56], [57] for comprehensive overviews. Even though SCA and FI countermeasures do not solve all problems, there is a sound understanding of attacks and countermeasures.

We focus on hardware Trojans to weaken security by manipulating the underlying hardware. To evaluate the severe consequences of low-level hardware manipulations at a larger scale, we obtained numerous publicly available, third-party Advanced Encryption Standard (AES) implementations from OpenCores and an NSA website to achieve variability. Each AES IP core provides an interface to set a key and encrypt or decrypt user data. To communicate with the environment, we extended each design to include a Universal Asynchronous Receiver Transmitter (UART)/RS-232 interface that can be used as part of an FPGA implementation. To be as close as possible to a practical scenario, we furthermore augmented each AES IP core with a self-test[1], see Section 5.1. Note that we made no changes to the underlying AES design, but merely integrated necessary hardware components with a full-fledged IP core.

SRAM-based FPGAs. In our offensive case studies, we focus on SRAM-based FPGAs. As noted in Section 2.4, the majority of currently deployed SRAM-based FPGAs from Xilinx and other vendors can be affected by post-deployment hardware Trojan injections since the bitstream is either not protected or the cryptographic protection can be circumvented by means of SCA attacks. Thus, an adversary can read-out and transform the proprietary bitstream file format to a readable gate-level netlist, perform reverse engineering and manipulation of security-critical design parts, and regenerate and deploy the bitstream.

Notation. We use the following notations: p - Plaintext (16 bytes), k - Key (16 bytes), $c = \text{AES}_k(p)$ - Ciphertext (16 bytes), $(p_{\text{ref}}, c_{\text{ref}})$ - Plaintext/ciphertext pair for the self-test, k_{st} - Key for the self-test, k_u - Key for user data.

5.1 Case Study: Disarm Cryptographic Self-Tests

Several works have demonstrated that targeted manipulation of third-party cryptographic hardware implementations have serious consequences, ranging from key leakage to surreptitiously weakened ciphers, even for high-security real-world devices, see [59], [47], [48], [49]. These attempts are usually detected by mandatory self-tests in cryptographic IP cores [58]. To utilize these powerful attacks for realistic IP cores, an adversary must disarm the self-test prior to the manipulation. In this case study, we demonstrate for the first time how the self-test circuitry can be both algorithmically reverse engineered and manipulated in a way that the aforementioned attacks can be performed.

Detailed System Model. We assume the following generic workflow for the cryptographic IP core:

1) Upon initialization, the IP core conducts a power-up self-test with an internally stored self-test key k_{st} and a reference plaintext p_{ref}.
2) The IP core checks whether the computed ciphertext is equal to the internally stored reference ciphertext, i.e. $c_{\text{ref}} \stackrel{?}{=} \text{AES}_{k_{\text{st}}}(p_{\text{ref}})$.
3) If the self-test is successful, a user key k_u is passed to the AES core and used to encrypt or decrypt user data. Otherwise, the cryptographic module enters an error state.

1. "A cryptographic module shall perform power-up self-tests and conditional self-tests to ensure that the module is functioning properly", see FIPS PUB 140-2 [58].

Device	Synthesis Option	AES Design 1	2	3	4	5	6	7	8	9	10	11	12	13
Spartan-3E XC3S1600E	area	●	●	n.a.	●	●	●	●	●	●	●	●	●	●
	balanced	●	●	n.a.	●	●	●	●	●	●	●	●	●	
	speed	●	●	n.a.	●	●	●	◐	●	●	●	●	●	
Spartan-3 XC3S1000	area	●	●	n.a.	●	●	●	n.a.	●	●	●	●	●	●
	balanced	●	●	n.a.	●	●	●	n.a.	●	●	●	●	●	●
	speed	●	●	n.a.	●	●	●	n.a.	●	●	●	●	●	●
Spartan-6 XC6SLX16	area	●	◐	●	●	◐	◐	n.a.	●	●	●	●	●	●
	balanced	●	●	●	●	●		n.a.	●	●	●	●	●	●
	speed	●	●	●	●	●		n.a.	●	●	●	●	●	●
Virtex-4 XC4VLX25	area	●	●	n.a.	●	●	●	n.a.	●	●	●	●	●	●
	balanced	●	●	n.a.	●	●	●	n.a.	●	●	●	●	●	●
	speed	●	●	n.a.	●	●	●	n.a.	●	●	●	●	●	●
Virtex-5 XC5VLX50	area	●	●	●	●	●	●	●	●	●	●	●	●	●
	balanced	●	●	●	●	●	●	●	●	●	●	●	●	●
	speed	●	●	●	●	●	●	●	●	●	●	●	●	●
Virtex-6 XC6VLX75	area	●	◐	●	●	◐	◐	●	●	●	●	●	●	●
	balanced	●	◐	●	●	●	●	●	●	●	●	●	●	●
	speed	●	◐	●	●	●	●	●	●	●	●	●	●	
7 series XC7K70T	area	●	◐	●	●	◐	◐	●	●	●	●	●	●	●
	balanced	●	◐	●	●	●	●	●	●	●	●	●	●	●
	speed	●	◐	●	●	●	●	●	●	●	●	●	●	●

Table 2
Evaluation results of self-test reverse engineering. ●: successfully reverse engineered, ◐: reverse engineering required minor manual netlist inspection, *n.a.*: the given AES could not be implemented for the device, blank: reverse engineering did not yield a result.

Note that we do not make assumptions about how the IP core is integrated into a larger system or whether the user key k_u is stored internally or supplied from the external environment.

Adversary's Goal. The high-level goal of the adversary is to perform targeted manipulation of the cryptographic computation to reveal the employed key k_u or weaken the cipher (e.g. by an S-box substitution [59], [47], [48], [49]). To perform a successful manipulation, the adversary is required to disarm the self-test circuit and trick it in a way that it always returns *successful* irregardless of the manipulated cryptographic implementation.

5.1.1 Algorithmic Reverse Engineering of Self-Test Circuits

To disarm and manipulate the self-test, the adversary has to first reverse engineer which gates and signals implement this functionality. To this end, we first detail how cryptographic self-tests are usually implemented and second we present our novel technique to determine how such structures can be automatically identified.

Cryptographic Power-Up Self-Tests. A self-test of a cryptographic module is usually realized as several additional states in the design's FSM that runs the cryptographic algorithm on some a priori computed, internally stored reference data. If the reference and the dynamically computed values are not equal, the design transitions into an *error state* (and does not perform any further operation), cf. [55].

Hereinafter, we describe the automatic reverse engineering strategies for two distinct device family series from Xilinx. In particular, we highlight how the different FPGA architectures (designed for 4-input LUTs or 6-input LUTs) affect our reverse engineering strategy. We want to emphasize that our search mainly focuses on the crucial 128-bit comparator which checks the equivalence of c_{ref} and the dynamically computed $\text{AES}_{k_{\text{st}}}(p_{\text{ref}})$.

Reverse Engineering. The general idea of our automated reverse engineering technique is to search for the comparator circuit that computes the equivalence of c_{ref} and $\text{AES}_{k_{\text{st}}}(p_{\text{ref}})$. Even though comparators can be realized by numerous FPGA gates (e.g, LUTs, multiplexers, AND, or carry gates), we developed a generic approach shown in Algorithm 2.

Algorithm 2 Self-Test Circuit Reverse Engineering

Input: D - Design gate-level netlist
Output: $\mathcal{S}$ - Set of self-test circuits
1: set $\mathcal{S} \leftarrow \emptyset$
2: list $\mathcal{L} \leftarrow \emptyset$
3: **for** gate $g \in \mathcal{D}$ **do**
4: **if** check_hamming_weight$(g) = true$ **then**
5: $\mathcal{L}$.append(g)
6: set of comparators $\mathcal{C} \leftarrow$ merge$(\mathcal{L})$
7: **return** $\mathcal{S} \leftarrow$ merge_comparators$(\mathcal{C})$

In lines ❸ - ❺, the function check_hamming_weight analyzes whether each gate's Boolean function of all *active* input pins (neither static GND nor VCC) implements a function that returns a distinctive output bit (e.g., logical 1 or logical 0 for exactly one input) and the complementary bit for all other active input pin assignments. We then analyze the neighborhood of each candidate gate and merge connected ones ❻. Therefore, we first check whether candidate gates are direct successors or predecessors of each other as well as whether the output of several candidate gates merge in

further target gates. We repeat this action until no further candidate or target gate can be added to the comparator. We then merge the identified comparators to determine whether they form a *multi-comparator* ❼, i.e. that checks more than one value. Note that this multi-comparator is not identified in ❻, since the gate that combines two comparators usually consists of a Boolean OR. Finally, we output all identified (multi-)comparators and output their respective bit-width which is derived by counting the number of different inputs.

With the assistance of HAL, we developed a plugin that yields a list of comparators and their corresponding width. Prior to the large-scale evaluation of the comparator detection algorithm, we briefly describe how such identified comparators can be manipulated.

5.1.2 Manipulation of a Self-Test

We present two implementation strategies to bypass a self-test: (1) always bypassed, and (2) conditionally bypassed.

Always Bypassed. To bypass the self-test for any input, we manipulate the final gate in the comparator which decides whether an input value matches the expected one. For example, if the final gate is a LUT, we change its configuration to always output a logical 1 (or 0 depending on what the comparator interprets as *true*). Otherwise, we simply alter each identified LUT in the comparator appropriately so that it always (erroneously) outputs *true*.

Conditionally Bypassed. If the comparator of the self-test is utilized by other circuitry, we can simply add a trigger condition to the design that checks whether the comparator is used for the self-test or not. Typically, this requires some additional reverse engineering of the design's control logic, see [29], [18].

5.1.3 Evaluation

For the large-scale evaluation of our proposed self-test detection, we target a range of Xilinx FPGA families and several synthesis options for each of the AES designs. Note that we employed the ATHENa framework [60] to automate this process of gathering diverse variants of each design.

Our large-scale analysis results are depicted in Table 2. Note that we targeted 128-bit comparators due to the state width of AES. The vast majority of the comparators were detected for the diverse FPGA families, synthesis options, and designs. In the cases marked with ◐, the comparator deviates more distinctively from 128 or the comparator was only found in two distinct parts. This occurred due to the fact that additional logic is performed by some LUTs in the comparator which complicates the algorithmic reverse engineering, however these cases can be easily resolved with manual analysis. In the small number of cases where the comparator reverse engineering was not successful, the self-test comparator was split up into more than two single-comparators or the recovered bit-width exceeded 128.

Practical Verification. To confirm that we manipulated the self-test circuit correctly, we performed an S-Box substitution attack [47] and verified the erroneous AES computation for the XC6SLX16 on a sample basis. Note that we used the always bypass manipulation. For the other FPGA families, we simulated the overall design's behavior and verified the successful manipulation of the different AES IP cores on a sample basis.

5.1.4 Discussion

We acknowledge that the self-test used for evaluation was implemented by us, however, to the best of our knowledge, there is no open-source available FIPS-certified AES implementation which includes a crucial self-test. Our approach is easy to adapt for different implementation structures and with HAL the identification process can successfully be automated with little effort.

5.2 Case Study: Wiretapping Keys in IP Cores

In contrast to the prior case study, we demonstrate that we are able to semi-automatically insert hardware Trojan circuits into an existing IP core to wiretap and leak utilized AES keys.

Detailed System Model. For this case study, we assume the same system model as described in Section 5.1.

Adversary's Goal. The primary goal is to insert a Trojan that leaks sensitive data while an AES core is actively used. Therefore, the adversary attempts to wiretap the entire state after the *SubBytes* transformation in the first round. Subsequently, the adversary aims to leak the sensitive data to the external environment to enable key recovery.

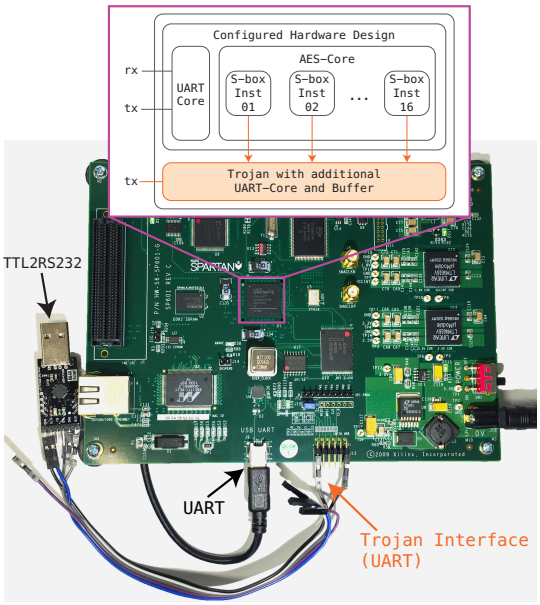


Figure 2. Xilinx SP601 development board used for experimentation. A wiretap Trojan was inserted into an existing low-level netlist. Sensitive data is leaked via an additional Trojan UART connected to a TTL2RS232 USB module.

5.2.1 Algorithmic Key Detection

Since the adversary intends to leak the state after the *SubBytes* transformation in the first round, we used HAL to perform the S-Box detection techniques described by Swierczynski *et al.* [47]. The algorithm outputs a list of S-Box instances and the bit order of the S-Box output. Note that the algorithm does not reveal any information regarding which state byte belongs to which S-Box instance so the correct permutation is unknown.

5.2.2 Key Recovery

In the following, we explain the key recovery for our AES core under attack. The core is a round-based implementation and therefore utilizes sixteen S-Box instances in the *SubBytes* step and four S-Box instances to implement the AES key schedule. Note that we confirmed this architecture by means of the number of identified S-Box instances. Furthermore, it is noteworthy that our procedure is not limited to this AES core, but with minor modifications applicable to other designs as well.

To gain knowledge of the correct byte permutation of the leaked 16-byte state, we analyzed which S-Box sub-circuit processes which byte of the state by means of simulation. Additionally, we observed the AES state over time. With this information we know which data buses are the relevant ones to be wiretapped and at what point in time we need to wiretap.

Once wiretapping is done, the retrieved data must be forwarded to the external environment. Therefore, we inserted a UART/RS-232 core and merged it with the given low-level netlist. The UART transmitter was connected to an unused I/O pin. To retrieve the wiretapping result we attached an additional TTL2RS232 USB receiver to this pin, cf. Figure 2. Once the wiretapped state is obtained, the adversary can easily compute the employed AES key by applying the inverse S-Box operation followed by an XOR with the plaintext.

5.2.3 Discussion

With HAL we are able to add any plain high-level IP core into an existing low-level design. Using the FPGA vendor's toolchain, a new bitstream can be generated to verify the functionality of the modified design. Although we only analyzed one specific core in this case study, our results are transferable to different designs as our technique is generic.

A more elegant technique of leaking the data would include the design and use of an antenna circuitry in the FPGA design [61]. As long as the user does not probe each I/O pin or the near-field, both leakage approaches remain relatively stealthy.

6 Discussion

Implications. Since hardware security covers a broad research landscape, HAL can be utilized to accompany and support various directions for offensive and defensive intents and purposes. For example, reverse engineering provides valuable information for security engineers and improves the understanding of SCA attacks and countermeasures since more information about a hardware design implementation facilitates a more fine-grained security evaluation. We show that hardware reverse engineering and joint manipulation can indeed be carried out for FPGAs and ASICs after the device or design has been tested, its code reviewed, and formally verified.

A common countermeasure against reverse engineering is obfuscation [26]. Although obfuscation increases the effort a reverse engineer has to invest, it generally does not hinder eventual success. Using HAL, de-obfuscation methods can be integrated into the analysis to include automated means for obfuscation removal. For example, in recent work Wallat *et al.* [35] demonstrated automated de-obfuscation of *opaque predicates* used for IP watermarks.

In general, one could argue that instead of reverse engineering and manipulating a design, it would be easier to completely replace it. However, this is not a realistic scenario since the reverse engineer does not know the exact design specifications beforehand. Thus, fully replacing the design without prior reverse engineering typically leads to a design which deviates from the original implementation or requires massive effort.

Other Vendors. Despite our case studies targeting Xilinx FPGAs, our research is not specific to Xilinx devices and can be adapted to devices from other FPGA vendors as well. For example, the bitstream encryption scheme of Intel's Stratix II and Stratix III SRAM-based FPGA families can also be circumvented by means of SCA attacks [42]. To the best of our knowledge, bitstream file format reverse engineering for these families has so far not been practically demonstrated, but we expect that this step can be conducted as well. Project IceStorm [46] demonstrated successful bitstream reverse engineering to a human-readable netlist for iCE40 FPGAs. In 2012, Skorobogatov *et al.* [62] demonstrated key extraction from a Actel/Microsemi ProASIC3 device. This action can result in the decryption and extraction of bitstream information.

Future Work. We plan to explore further (semi-) automatic reverse engineering techniques and evaluate their capabilities for ASIC and FPGA designs. The more that is publicly known about what information can be algorithmically disclosed, the better sound threat estimation and countermeasure development can be performed. Additionally, we plan to address the big picture: examination of how human reverse engineers make sense of hardware designs [27]. Once an understanding of this process is revealed, obfuscation techniques can be designed which target not only automatic reverse engineering techniques, but also the thought processes of the human analyst.

7 Conclusion

Hardware Trojans have become a major threat for today's systems and applications. For both offensive and defensive research in the hardware Trojan area, the reverse engineering of high-level information from low-level placed-and-routed netlists is indispensable (e.g., to disclose security-critical circuitry, to inject Trojans, or to disclose the presence of malicious circuitry).

In this work, we closed several important research gaps: First, we introduced our generic netlist reverse engineering and manipulation framework HAL. The framework allows for the development of custom tools to automate time-consuming and complex reverse engineering. Second, we presented our hardware Trojan detection technique ANGEL which is based on Boolean function analysis and graph neighborhood analysis. The algorithm was implemented as a plugin for HAL. Third, we demonstrated the manifold applicability of HAL in a variety of case studies that focus on the real-world threat posed by hardware Trojans in cryptographic hardware designs. We extended the adversary tool arsenal to demonstrate how to automatically invalidate cryptographic self-tests and wiretap cryptographic key circuitry.

More importantly, we demonstrated that the development of automated custom tools for reverse engineering and hardware Trojan injection is not as challenging and time-consuming as previously thought, i.e. the time needed to reverse engineer and surreptitiously weaken a design is only several hours with the help of HAL.

Since we believe that our work raises awareness of a real-world attacker's capabilities, HAL represents a fundamental building block for future research. We plan to publicly release HAL and our analysis plugins, such as ANGEL, to the research community.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We also thank all of our students working on and with HAL, in particular, Adrian Drees and Sebastian Maaßen. Part of this work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 Research and Innovation programme (ERC Advanced Grant No. 695022 (EPoCH)), and National Science Foundation (NSF) awards CNS-1563829 and CNS-1318497.

REFERENCES

[1] M. Rostami *et al.*, "A Primer on Hardware Security: Models, Methods, and Metrics," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1283–1295, 2014.
[2] A. Moradi *et al.*, "On the Vulnerability of FPGA Bitstream Encryption against Power Analysis Attacks: Extracting Keys from Xilinx Virtex-II FPGAs," in *ACM CCS*, 2011, pp. 111–124.
[3] ——, "Side-Channel Attacks on the Bitstream Encryption Mechanism of Altera Stratix II," in *ACM/SIGDA FPGA*, 2013, pp. 91–100.
[4] P. Swierczynski *et al.*, "Interdiction in Practice—Hardware Trojan against a High-Security USB Flash Drive," *Journal of Cryptographic Engineering*, pp. 1–13, 2016.
[5] S. Bhunia *et al.*, "Hardware Trojan Attacks: Threat Analysis and Countermeasures," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1229–1247, 2014.
[6] S. T. King *et al.*, "Designing and Implementing Malicious Hardware," in *USENIX LEET*, 2008, pp. 1–8.
[7] L. Lin *et al.*, "Trojan Side-Channels: Lightweight Hardware Trojans through Side-Channel Engineering," in *CHES*. Springer, 2009, pp. 382–395.
[8] J. Rajendran *et al.*, "Blue team red team approach to hardware trust assessment," in *IEEE ICCD*, 2011, pp. 285–288.
[9] G. T. Becker *et al.*, "Stealthy Dopant-level Hardware Trojans," in *CHES*. Springer, 2013, pp. 197–214.
[10] J. Zhang *et al.*, "DeTrust: Defeating Hardware Trust Verification with Stealthy Implicitly-Triggered Hardware Trojans," in *ACM CCS*, 2014, pp. 153–166.
[11] S. Ghandali *et al.*, "A Design Methodology for Stealthy Parametric Trojans and Its Application to Bug Attacks," in *CHES*. Springer, 2016, pp. 625–647.
[12] K. Yang *et al.*, "A2: Analog Malicious Hardware," in *IEEE Symposium on Security and Privacy*, 2016, pp. 18–37.
[13] J. Rajendran *et al.*, "Security Analysis of Integrated Circuit Camouflaging," in *ACM CCS*, 2013, pp. 709–720.
[14] A. Vijayakumar *et al.*, "Physical Design Obfuscation of Hardware: A Comprehensive Investigation of Device and Logic-Level Techniques," *IEEE Trans. Information Forensics and Security*, vol. 12, no. 1, pp. 64–77, 2017.
[15] R. Torrance, "The State-of-the-Art in IC Reverse Engineering," in *CHES*. Springer, 2009, pp. 363–381.
[16] Texplained, https://www.texplained.com/process, [Online; accessed 19-May-2017].
[17] P. Subramanyan *et al.*, "Reverse Engineering Digital Circuits Using Structural and Functional Analyses," *IEEE Trans. Emerging Topics Comput.*, vol. 2, no. 1, pp. 63–80, 2014.
[18] T. Meade *et al.*, "Netlist Reverse Engineering for High-Level Functionality Reconstruction," in *ASP-DAC*, 2016, pp. 655–660.
[19] ——, "Gate-Level Netlist Reverse Engineering Tool Set for Functionality Recovery and Malicious Logic Detection," *International Symposium for Testing and Failure Analysis (ISTFA)*, 2016.
[20] A. Waksman *et al.*, "FANCI: Identification of Stealthy Malicious Logic using Boolean Functional Analysis," in *ACM CCS*, 2013, pp. 697–708.
[21] S. K. Haider *et al.*, "Advancing the State-of-the-Art in Hardware Trojans Detection," *IEEE Trans. Dependable and Secure Computing*, vol. PP, no. 99, pp. 1–1, 2017.
[22] S. E. Quadir *et al.*, "A Survey on Chip to System Reverse Engineering," *JETC*, vol. 13, no. 1, pp. 1–34, 2016.
[23] K. D. Pham *et al.*, "BITMAN: A Tool and API for FPGA Bitstream Manipulations," in *DATE*, 2017, pp. 894–897.
[24] Y. Alkabani *et al.*, "Active Hardware Metering for Intellectual Property Protection and Security," in *USENIX Security Symposium*, 2007.
[25] R. S. Chakraborty *et al.*, "HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection," *IEEE Trans. CAD of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1493–1502, 2009.
[26] B. Shakya *et al.*, *Introduction to Hardware Obfuscation: Motivation, Methods and Evaluation*. Springer, 2017, pp. 3–32.
[27] M. Fyrbiak *et al.*, "Hardware Reverse Engineering: Overview and Open Challenges," in *IVSW*, 2017.
[28] M. C. Hansen *et al.*, "Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.
[29] Y. Shi *et al.*, "A Highly Efficient Method for Extracting FSMs from Flattened Gate-Level Netlist," in *ISCAS*, 2010, pp. 2610–2613.
[30] ——, "Extracting Functional Modules from Flattened Gate-Level Netlist," in *ISCIT*, 2012, pp. 538–543.
[31] W. Li *et al.*, "Reverse Engineering Circuits Using Behavioral Pattern Mining," in *IEEE HOST*, 2012, pp. 83–88.
[32] ——, "WordRev: Finding Word-Level Structures in a Sea of Bit-level Gates," in *IEEE HOST*, 2013, pp. 67–74.
[33] P. Subramanyan *et al.*, "Reverse Engineering Digital Circuits Using Functional Analysis," in *DATE*, 2013, pp. 1277–1280.
[34] A. Gascón *et al.*, "Template-based circuit understanding," in *FMCAD*, 2014, pp. 83–90.
[35] S. Wallat *et al.*, "A Look at the Dark Side of Hardware Reverse Engineering – A Case Study," in *IVSW*, 2017.
[36] D. S. B. W. DC, "Report of the Defense Science Board Task Force on High Performance Microchip Supply," 2005.
[37] M. Tehranipoor *et al.*, "A Survey of Hardware Trojan Taxonomy and Detection," *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.
[38] M. Hicks *et al.*, "Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically," in *IEEE Symposium on Security and Privacy*, 2010, pp. 159–172.
[39] J. Zhang *et al.*, "VeriTrust: Verification for Hardware Trust," in *ACM/EDAC/IEEE (DAC)*, 2013, pp. 1–8.
[40] B. Cakır *et al.*, "Hardware Trojan Detection for Gate-level ICs Using Signal Correlation Based Clustering," in *DATE*, 2015, pp. 471–476.
[41] H. Salmani, "COTD: Reference-Free Hardware Trojan Detection and Recovery Based on Controllability and Observability in Gate-Level Netlist," *IEEE TIFS*, vol. 12, no. 2, pp. 338–350, 2017.
[42] P. Swierczynski *et al.*, "Physical Security Evaluation of the Bitstream Encryption Mechanism of Altera Stratix II and Stratix III FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 4, pp. 1–23, 2014.
[43] J.-B. Note and É. Rannaud, "From the bitstream to the netlist," in *ACM FPGA*, 2008, pp. 264–264.
[44] F. Benz *et al.*, "BIL: A tool-chain for bitstream reverse-engineering," in *IEEE FPL*, 2012, pp. 735–738.
[45] Z. Ding *et al.*, "Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation," *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 37, no. 3, pp. 299–312, 2013.
[46] C. Wolf *et al.*, "Project IceStorm," 2015.
[47] P. Swierczynski *et al.*, "FPGA Trojans Through Detecting and Weakening of Cryptographic Primitives," *IEEE TCAD*, vol. 34, no. 8, pp. 1236–1249, 2015.
[48] A. C. Aldaya *et al.*, "AES T-Box tampering attack," *J. Cryptographic Engineering*, vol. 6, no. 1, pp. 31–48, 2016.
[49] P. Swierczynski *et al.*, "Bitstream Fault Injections (BiFI) – Automated Fault Attacks against SRAM-based FPGAs," *IEEE TC*, vol. PP, no. 99, pp. 1–1, 2017.
[50] H. Salmani *et al.*, "On Design vulnerability analysis and trust benchmarks development," in *IEEE ICCD*, 2013, pp. 471–474.

[51] W. Snyder, "Verilator," https://www.veripool.org/wiki/verilator, [Online; accessed 19-May-2017].
[52] J. Lind-Nielsen, "BuDDy: A Binary Decision Diagram library," http://buddy.sourceforge.net, [Online; accessed 19-May-2017].
[53] "GitHub pybind11," https://github.com/pybind/pybind11, [Online; accessed 19-May-2017].
[54] A. Waksman *et al.*, "A red team/blue team assessment of functional analysis methods for malicious circuit identification," in *ACM/EDAC/IEEE DAC*, 2014, pp. 1–4.
[55] NIST, "Suite B Cryptography," 2001.
[56] S. Trimberger *et al.*, "FPGA Security: Motivations, Features, and Applications," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1248–1265, 2014.
[57] B. Badrignans *et al.*, *Security Trends for FPGAS: From Secured to Secure Reconfigurable Systems*, 1st ed. Springer, 2011.
[58] NIST, "Security Requirements for Cryptographic Modules," 2001.
[59] T. Kerins *et al.*, "A Cautionary Note on Weak Implementations of Block Ciphers," in *Workshop on Information and System Security*, 2006, p. 12.
[60] K. Gaj *et al.*, "ATHENa - Automated Tool for Hardware EvaluatioN," in *FPL*, 2010, pp. 414–421.
[61] J. Couch *et al.*, "An Analysis of Implanted Antennas in Xilinx FPGAs," in *ReConFig*, 2011, pp. 1–6.
[62] S. Skorobogatov *et al.*, "In the blink of an eye: There goes your AES key," *IACR Cryptology ePrint Archive*, pp. 1–7, 2012.

Marc Fyrbiak received his B.Sc. degree in computer science from TU Braunschweig, Germany in 2012 and his M.Sc. degree in IT security from Ruhr-Universität Bochum, Germany in 2014. He is currently working towards the Ph.D. degree at the Chair for Embedded Security, under the supervision of C. Paar. His research interests include reverse engineering of hardware and software systems, as well as security analysis of real-world devices.

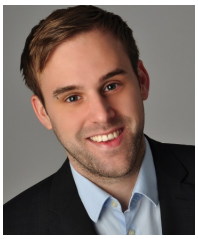
Sebastian Wallat received his B.Sc. degree in computer science from University Duisburg-Essen, Germany in 2012 and his M.Sc degree in IT security from Ruhr-Universität Bochum, Germany in 2016. He is currently working towards the Ph.D degree at the University of Massachusetts, Amherst, USA under the supervision of C. Paar. His research interests include malicious hardware Trojan design strategies to explore mitigation techniques, as well as reverse engineering of hardware and software.

Pawel Swierczynski received the B.Sc. and M.Sc. degrees in IT-Security from Ruhr-Universität Bochum, Germany, in 2010 and 2013, and was working towards his Ph.D. degree at the Chair for Embedded Security, under the supervision of C. Paar. His research was focused on the physical security of FPGAs as well as on practical attacks on real-world devices with a special emphasis on cryptographic FPGA design manipulation.

Max Hoffmann studied IT-Security at Ruhr-Universität Bochum, Germany. He finished his B.Sc. in 2015 and M.Sc. in 2017. Since he started his Ph.D. studies at the Chair for Embedded Security of C.Paar in 2016, his research has focused on hardware reverse engineering and obfuscation techniques.

Sebastian Hoppach received his M.Sc. degree in IT security from Ruhr-Universität Bochum, Germany in 2017. His research interests focus on the security of embedded hardware and software, including hardware reverse engineering and side-channel attacks. He is currently working as a security consultant.

Matthias Wilhelm finished his B.Sc in IT-Security from Ruhr-Universität Bochum in 2015. He is currently working towards a M.Sc degree from the same university. His primary research interest include reverse engineering of embedded devices including FPGAs.



Tobias Weidlich received his B.Sc. degree in IT security from Ruhr-Universität Bochum, Germany in 2014 and his M.Sc. degree in IT security from Ruhr-Universität Bochum, Germany in 2016. He is currently working as an IT security specialist for the IHK-CERT.

Russell Tessier (M'00-SM'07) received the B.S. degree in computer and systems engineering from Rensselaer Polytechnic Institute, Troy, NY, USA, in 1989, and the S.M. and Ph.D. degrees in electrical engineering from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 1992 and 1999, respectively. He is currently Professor of Electrical and Computer Engineering with the University of Massachusetts, Amherst, MA. His current research interests include computer architecture and FPGAs.

Christof Paar (Fellow, IEEE) received the M.Sc. degree from the University of Siegen and the Ph.D. degree from the Institute for Experimental Mathematics at the University of Essen, Germany. He holds the Chair for Embedded Security at Ruhr-Universität Bochum, Bochum, Germany, and is an Affiliated Professor at the University of Massachusetts Amherst, Amherst, MA, USA. His research interests include highly efficient software and hardware realizations of cryptography, physical security, security evaluation of real-world systems, and cryptanalytical hardware.