IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 29, 2019, accepted February 16, 2019, date of publication February 27, 2019, date of current version April 5, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2901949

A Comprehensive FPGA Reverse Engineering Tool-Chain: From Bitstream to RTL Code

TAO ZHANG, JIAN WANG, (Member, IEEE), SHIZE GUO, AND ZHE CHEN
School of Information and Communication Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China
Corresponding author: Jian Wang (wangjian3630@uestc.edu.cn)

This work was supported by the National Natural Science Foundation of China under Grant 61671110.

ABSTRACT As recently studied, field-programmable gate arrays (FPGAs) suffer from growing Hardware Trojan (HT) attacks, and many techniques, e.g., register-transfer level (RTL) code-based analyzing, have been presented to detect HTs on FPGAs. However, for most of the FPGA end users, they can only obtain bitstream, rather than the RTL code. Therefore, we present a new FPGA reverse engineering tool-chain. It can precisely transform the FPGA bitstream to an RTL code and therefore assists in HT detection. In detail, we first construct an integrated database involving the FPGA architecture information and the bitstream mapping information. Then, we build two tools, namely, bitstream reversal tool (BRT) and netlist reversal tool (NRT). They can be combined together to retrieve the RTL code from the FPGA bitstream in moderate time. To demonstrate the effectiveness of our tool-chain, we evaluate it qualitatively and quantitatively by using two benchmarks (ISCAS'85 and ISCAS'89) and three real applications (8051 core, 68HC08, and AES). Our tool-chain is comprehensive since it covers all the reverse engineering stages, from bitstream to netlist and from netlist to code, without any support from other tools. Moreover, it rebuilds the netlist with a 100% correct rate and retrieves RTL code, which is exactly, functionally equivalent to the original one for all our benchmarks. To the best of our knowledge, it is the first tool that can perform integrated, precise reverse engineering for FPGAs, paving the way for the netlist-/code-based HT detection.

INDEX TERMS FPGA, reverse engineering, bitstream, hardware trojan.

I. INTRODUCTION

Nowadays, FPGAs are widely used in various domains due to their high performance and reconfigurability, such as aerospace systems, automotive, high-performance computing and IoT (Internet of Things), etc., as shown in Fig. 1a, and therefore, attract growing attentions from hardware hackers [1]. Recently, some adversaries successfully inserted HTs (Hardware Trojans) into FPGAs, aiming to steal confidential data like encryption keys or change their original functionality. For example, Chakraborty *et al.* [2] searched for the empty space of FPGA bitstream and then implanted HTs in it. Fern *et al.* [3] presented a novel HT and hid it in the RTL code. Besides, other phases during FPGA lifetime, such as device design, fabrication and assembly, etc., can also be exploited for HT inserting, as shown in Fig. 1b [4]. Such circumstances greatly challenge the FPGA security, calling for more powerful HT detection techniques.

FPGA HT detection can be divided into two categories. One is based on physical information analyzing which is with high efficiency in detecting the large and medium scale Trojans in FPGAs by analyzing their side channel information, e.g., Iwase *et al.* [5] isolate the chip power signature to detect HT in the Trojan-infected FPGAs by using SVM (Support Vector Machine). The other refers to design file analyzing method which can isolate the Trojan-infected FPGAs by analyzing the characteristics of their netlist or

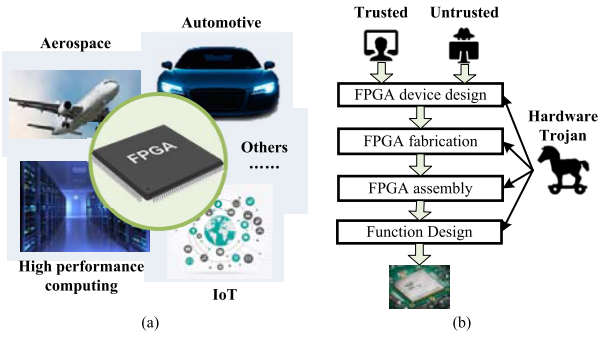


FIGURE 1. The background of FPGA hardware trojan detection. (a) FPGA applications in various domains. (b) Possible phases for Hardware Trojans insertion in FPGA lifetime.

The associate editor coordinating the review of this manuscript and approving it for publication was Remigiusz Wisniewski.

RTL code. This method, albeit often time-consuming, can find out small and even tiny Trojans in FPGAs. For example, Hicks *et al.* [6] propose a UCI (Unused Circuit Identification) method which identifies the never used RTL code, shrinking the code space left for HTs. Guo *et al.* [7] perform theorem proving and equivalence checking on RTL code to ensure there are no additional functions which are not defined in design specification. Zhang *et al.* [8] detect HTs in a design by identifying the redundant logics in its netlist. However, common FPGA end users can only access FPGA bitstream[1] rather than netlist or RTL code. To address this dilemma, we propose a new comprehensive FPGA reverse engineering tool-chain, which can precisely convert bitstream to RTL code, paving the way for netlist/code-based hardware Trojans detection.

In this paper, our main contributions are threefold:

- We propose a tool-chain for FPGA reverse engineering, involving three tools, namely, *library generator* (LG), *bitstream reversal tool* (BRT) and *netlist reversal tool* (NRT). LG individually tests all kinds of FPGA resources, removes the interferences among different components, and thereby, obtains an exhaustive and accurate mapping relationship between bit values and FPGA configuration. BRT retrieves a netlist from the intercepted bitstream. To get a precise netlist, we embed a patching algorithm in BRT to deduce the working states of some special elements which depend on not only the value of configuration bits but also the state of neighbor elements. NRT recovers RTL code from netlist. To ensure the correctness of decompiling process, it splits the retrieved netlist into multiple independent clusters and replaces their elements and connections with functionally equivalent code.
- We perform extensive experiments to evaluate our tool-chain qualitatively and quantitatively, and make comparison with some existing re-engineering tools. In qualitative evaluation, we illustrate that we construct an integrated tool-chain which covers all reversing stages, i.e., bitstream-to-netlist and netlist-to-code, outperforming the existing tools. In quantitatively evaluation, we report the correct rate of decompiled netlist, i.e., 100%, for 13 benchmarks and validate that their recovered RTL code is functionally equivalent to the original one. To the best of our knowledge, it is the first integrated and accurate re-engineering tool which can decompile FPGA bitstream to code without any support from other tools.
- We present a case study to demonstrate the effectiveness of our tool-chain in assisting HT detection. In detail, we first recover the netlist and RTL code from bitstream which corresponds to a Trojan-infected application by using our tool-chain, and then employ two HT detection techniques, COTD (Controllability and Observability for hardware Trojan Detection) [9] and CA (Coverage Analysis) [10], to check the recovered files, respectively. The experimental results reveal that the malicious logics can be successfully identified in this working flow. Hence, our tool-chain is helpful for ensuring FPGA hardware security.

[1]FPGA bitstream is a binary file that contains configuration data for an FPGA. Its detailed format is usually proprietary and undocumented.

The rest of this paper is organized as follows. In Section II, we briefly discuss related work. In Section III, we detail our tool-chain. The methodology on how to construct bitstream mapping database and the working flow of our BRT and NRT tools are introduced. In Section IV, we evaluate our tool qualitatively and quantitatively, and in Section V, we present a case study to demonstrate the effectiveness of our tool-chain in assisting HT detection. Finally, we conclude the paper in Section VI.

II. RELATED WORK

Hardware reverse engineering is a process of understanding the architecture and functionality of electronic systems via special methods. At chip level, it touches upon two kinds of devices, ASIC (Application-Specific Integrated Circuit) and FPGA.

A. ASIC REVERSE ENGINEERING

The purpose of ASIC reverse engineering is to obtain high-level description of the manufactured circuits. It contains two stages: chip reverse engineering and netlist reverse engineering. Chip reversing recovers netlist from the implemented circuit, involving both *non-destructive* and *destructive* techniques [11]. In non-destructive methods, X-ray tomography is often used to provide layer images for chips. For example, the work in [12] takes advantage of a beam of X-rays at a piece of Intel processor to reconstruct the chip transistors and wires. Destructive reverse engineering, on the other hand, refers to etching and grinding chip layer-by-layer, which is quite time-consuming and requires expensive instruments [13]. As such, only several companies can offer destructive reverse engineering services.

Netlist-level reverse engineering generates high-level description from the gate-level netlist, and plenty of techniques have been reported. For example, Li *et al.* [14] present a systemic way to automatically derive code structure from circuit netlist. Meade *et al.* [15] propose a netlist reverse engineering tool-set which can recover the functionality of the netlist into FSMs (Finite State Machines). Subramanyan *et al.* [16] identify all components from the netlist, such as register, counter and adders, etc., and then describe them by using high-level abstraction. Thereafter, they improve the efficiency of the method, scaling it to large-scale ICs with almost million elements [17].

B. FPGA REVERSE ENGINEERING

FPGA reverse engineering aims to retrieve high-level description from the bitstream. It can be divided into two phases, *bitstream reversing* and *netlist reversing*.

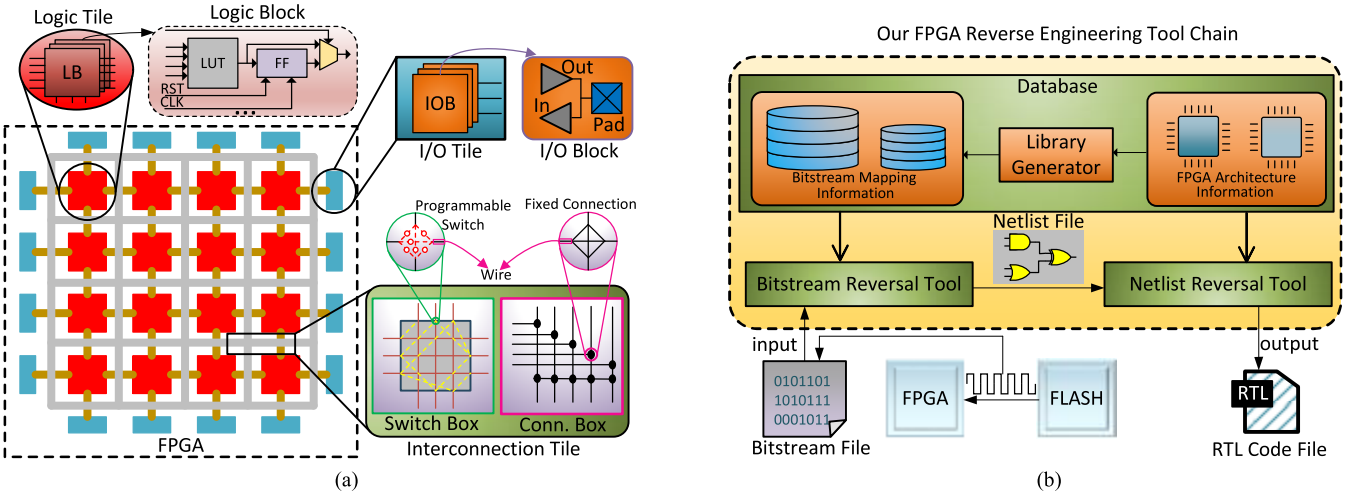


FIGURE 2. An overview of (a) FPGA architecture and (b) reverse engineering flow.

Bitstream reversing decompiles FPGA bitstream to netlist. As a pioneer work of this domain, Ziener *et al.* [18] investigate the bitstream of Virtex-II FPGA and abstract the LUT (Look Up Table) contents from it. Afterwards, Note and Rannaud [19] propose a cross-correlation algorithm to build the relationship between the bitstream and the configurable elements for multiple Xilinx FPGAs. By using the same algorithm as [19], Benz *et al.* [20] present a tool called BIL, which considers both configurable and unconfigurable FPGA elements, and successfully recovers a fraction of netlist from bitstream. After that, Yoon *et al.* [21] correct some errors in BIL to improve its reversing accuracy, generating an advanced tool to retrieve netlist from FPGA bitstream. To improve the reversing accuracy, Ding *et al.* propose two new algorithms, PK (Position Known) analysis and PUK (Position Un-Known) analysis, to collect the bitstream mapping information. The experiments reveal that the correctness of their reversed netlists can achieve at least 88% [22]. In [23], the project IceStorm demonstrates a reversing tool which can successfully decompile the bitstreams of Lattice iCE40 FPGAs. In contrast to the above reversing tools focusing on unencrypted bitstreams, some works move their targets on the encrypted files. For example, Swierczynski *et al.* [24] propose a novel attack named BiFI, which can manipulate an encrypted bitstream to obtain valuable information from it without the need of full bitstream reversal.

FPGA netlist reversing generates high-level description, e.g., RTL code, based on the netlist file. Since FPGA netlist is similar to ASIC netlist, the netlist reversing techniques for ASIC can be easily implanted to FPGA. Cheremisinov [25] puts his efforts in this domain. For Xilinx FPGAs, he first convert netlist to a readable XDL (Xilinx Design Language) file, and then design an automation tool to generate HDL (Hardware Description Language) code according to it.

III. OUR WORK

In this section, we give an overview of FPGA architecture, detail our FPGA reverse engineering tool-chain, and report the time complexity of our tool-chain.

A. FPGA ARCHITECTURE OVERVIEW

Fig. 2a gives an overview of the FPGA architecture, which consists of three parts, namely, logic tile, I/O (Input/Output) tile and interconnection tile.

Each logic tile contains multiple LBs (Logic Blocks) with various functional elements, such as LUTs, FFs (Flip-Flops) and multiplexers, etc. These elements can be configured through bitstream to implement the desired functions.[2] I/O tiles are located at the periphery of FPGA, they work as communication interface bridging the FPGA to the outside world. Similar to the logic tile, each I/O tile contains multiple IOBs (I/O Blocks) as well. In one IOB, there are several configurable elements, e.g., inverter, buffer and pull-up resistance, etc., and one PAD, a non-configurable element. By using bitstream, IOBs can be set to either input or output mode. Interconnection tile is the internal communication infrastructure of FPGA, transferring signals between LBs and IOBs. Two kinds of boxes, configurable switch box and unconfigurable connection box, are involved in each interconnection tile. Switch box contains plenty of PS (Programmable Switches) which can be configured to be ON or OFF state, building connections between wires accordingly. On the other hand, connection box links specific wires together through a set of *fixed* connections.

Therefore, FPGA physical structure can be easily modified by programming the LBs, IOBs and switch boxes with the bitstream, implementing various functions as we want.

[2]Note that some other papers use 'element', 'logic element' and 'logic cell' interchangeably. In this paper, we use the word 'element' throughout the paper.

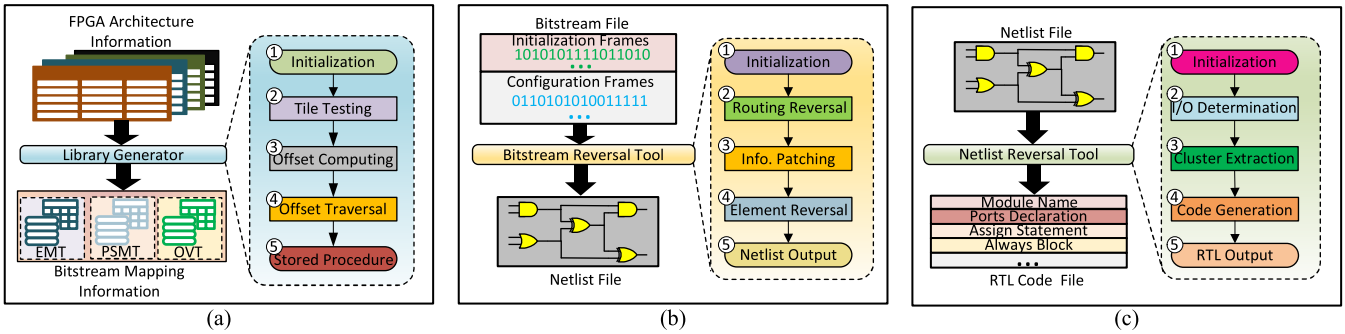


FIGURE 3. The details of FPGA reverse engineering tool-chain. (a) Library generator. (b) Bitstream reversal tool. (c) Netlist reversal tool.

B. OUR FPGA REVERSE ENGINEERING TOOL-CHAIN

The flow of our FPGA reverse engineering is given in Fig. 2b. First, we wire-tap the configuration lines on PCB (Printed Circuit Board) and power on our target FPGA. When the FPGA loads its bitstream from the external non-volatile memory, we immediately intercept the bitstream and save it. Then, we feed the obtained bitstream into our tool-chain. Supported by the database which contains FPGA architecture information and bitstream mapping information, our BRT (Bitstream Reversal Tool) decompiles the bitstream to the netlist, and subsequently, our NRT (Netlist Reversal Tool) converts the netlist to RTL code. Note that unless otherwise stated, the FPGA in this paper refers to SRAM-based FPGA, rather than antifuse and flash FPGAs which have no needs for outside bitstream. Moreover, the bitstream fed into our tool-chain should be an unencrypted file.[3]

1) DATABASE

Database is the bedrock for FPGA reverse engineering. It supplies two parts of necessary information for our BRT and NRT tools, referring to FPGA architecture and bitstream mapping.

FPGA architecture information includes all hardware details of our target FPGAs, such as the quantity, location and all possible working patterns for functional elements and their connections. This information can be obtained from the corresponding FPGA vendor. For example, the Xilinx ISE (Integrated Synthesis Environment) provides a set of *textual* files describing the architecture of Xilinx FPGAs separately. Bitstream mapping information points out the relationship between the value of bits and the working patterns of configurable components, i.e., the logic states of elements or the ON/OFF status of programmable switches. We integrate a library generator in our database to figure out the mapping information through thorough black-box testing.

The working flow of our library generator is shown in Fig. 3a.

[3]Most of low-end FPGAs, e.g., Xilinx Spartan-6 6SLX25 and Intel Cyclone IV EP4CE6, provide unencrypted bitstreams which can be directly decompiled by our tool-chain. For the encrypted bitstreams of high-end FPGAs, they should be decrypted through some advanced techniques, e.g., side channel attack [26], [27], before our tool-chain decompiles them.

Step 1 (Initialization): We first directly write a blank netlist file N_0 and convert it to a bitstream $\mathbf{B}_0$. Note that a bitstream includes both initialization frames and configuration frames. The initialization frames set the booting modes of FPGA while the configuration frames determine the implemented function in FPGA. In this paper, our library generator only focuses on the configuration frames since we aim to retrieve the FPGA function.

Step 2 (Tile Testing): For all configurable elements in a logic tile, we add them into N_0, one element at a time, and test their mapping information. In detail, we first build a tabulation to record the possible working patterns for the added element, and then go through its possible working patterns one-by-one to generate a group of netlists. Afterwards, we transform these netlists to bitstreams and further compare them with $\mathbf{B}_0$ separately. In this way, we can find out the bits with different values and further determine the mapping relationship for the added element. Note that 1) the mapping relationship for components in I/O and interconnection tiles can be determined similarly, and 2) the tile testing time is linearly proportional to the number of components in a tile since we tackle every component individually.

Step 3 (Offset Computing): Offset value is the distance of configuration frames from one tile to another, as defined in [22]. In this step, we find out the offset value of two tiles with the same type, i.e., two logic tiles or two I/O tiles or two interconnection tiles. Specifically, we first choose two logic tiles, one tested tile as our baseline and one untested tile as our target, set them with the same configuration, and modify the netlist N_0 accordingly. In other words, we separately modify the blank netlist to implement the same function on the baseline tile and the untested tile. As such, we can obtain two modified netlists from N_0, and then generate two bitstream files for calculating the offset value. Fig. 4 gives an example to illustrate this procedure intuitively. As we can see, the tested tile is configured through the frame i to $i+N$, while the untested tile, e.g., Tile_1, corresponds to another N frames from j to $j+N$. The distance between frame i to frame j is exactly the offset value for Tile_1 against the baseline, i.e., $offset_1 = j - i$. For the other two kinds of tiles (I/O and Interconnection), we repeat the above operations and get their offset values separately.

Step 4 (Offset Traversal): We move our target to another untested tile in the same type while maintaining the baseline unchanged. Then, we find out the offset value between the new target and the baseline. By repeatedly performing the above operations, we can figure out all offset values for untested tiles, as shown in Fig. 4. After that, we can deduce all mapping information for the target FPGA by using the baseline mapping relationship and the offset values.

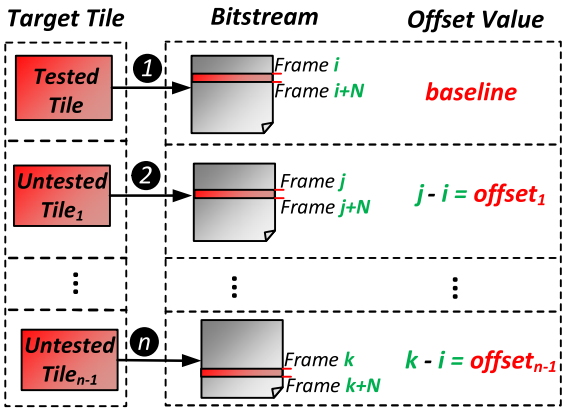


FIGURE 4. The procedure of offset computing and traversal.

Step 5 (Stored Procedure): For convenience, we save the mapping information in three pre-defined tables, namely EMT (Element Mapping Table), PSMT (Programmable Switch Mapping Table) and OVT (Offset Values Table). EMT contains the mapping relationship of elements in the baseline logic and I/O tiles, while PSMT refers to that of switches in the baseline interconnection tile. Note that EMT and PSMT has the same structure, they are multiple-row tables in which each row consists of four parts, a component's i) name, ii) one possible working pattern, iii) the configuration bits addresses, and iv) the bits' value corresponding to the pattern. On the other hand, OVT records the offset values for other tiles against the baseline. It is also a multiple-row table in which each row comprises three parts, i.e., a tile's i) name, ii) type, and iii) offset value.

While presenting the above steps, we make the following notes.

1. In order to get accurate mapping information, we perform thorough black-box testing for FPGA resources in a one-by-one manner. This sometimes disobeys the FPGA design rules as some functional elements and programmable switches in FPGA cannot be used alone. Therefore, we disable the DRC (Design Rule Check) function in EDA tool before we generate bitstream from the modified netlists.[4]

2. Although we collect the solely mapping relationship from configurable resources to bit values, we still come across the MVOE problem (Multiple Values for One Element) [22] which means that the same value of configuration bits may be mapped to multiple working patterns. To determine the working pattern of the elements with MVOE problem, we need the patterns of their neighbors in addition to mapping relationship. Hence, we propose a patching method and implement it in our BRT to address this issue. More details are discussed in Section III-B2.

3. We use offset values to deduce the mapping information for tiles except the three baselines which corresponds to I/O, logic and interconnection tiles. This method is reasonable due to the high regularity of FPGA architecture, as discussed in previous work [19], [22]. In addition, compared to the time-consuming testing for elements and connections in a one-by-one manner, offset computing can be quickly finished to figure out the mapping relationship for a tile.

[4]Most of the FPGA vendors integrate DRC function in their EDA (Electronics Design Automation) tools to monitor the behavior of netlists. Once a netlist disobeys the predefined rules, DRC generates a warning information and hinder the conversion from netlist to bitstream.

2) BITSTREAM REVERSAL TOOL

BRT transforms a bitstream to a netlist file which details the function of used elements and the routing paths among them. The working flow of BRT is drawn in Fig. 3b.

Step 1 (Initialization): In this step, we read the intercepted bitstream file and load the necessary information, i.e., the bitstream mapping and FPGA architecture information in our database.

Step 2 (Routing Reversal): According to the bitstream and database, we find out the turn-on programmable switches, and separately take them as a starting point in BFS (Breadth-First Search) algorithm to recover all signal nets [28]. Here net is composed of a set of connections which transfer the same signal from a source element to one or more sink elements.

Step 3 (Information Patching): As stated in Section B1, we test the FPGA elements one-by-one and encounter a MVOE problem, which means that multiple working patterns may be mapped into the same configuration bits value. By investigating the working mechanism of FPGA elements, we find that the independent testing method, albeit easy to be implemented, may result in information loss when it decouples the elements.

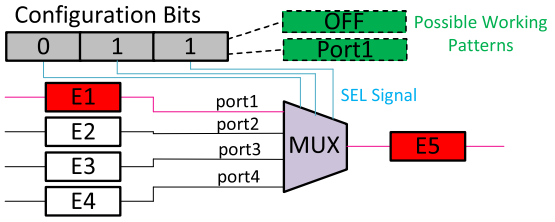


FIGURE 5. An example of MVOE problem.

For example, the working pattern of MUX element in Fig. 5 is determined through three configuration bits (SEL signal) in combination with a driven signal from upstream elements. When the three configuration bits are set to be '011', the MUX may be either powered OFF (no driven signal from E1) or in switching status (from

the first input port to its output port when E1 is working). Therefore, we cannot determine the MUX working pattern if we only access its configuration information. To solve this problem, we pick out all elements with uncertain patterns, analyze their correlation with neighbors and patch the missed information to elements accordingly.

Algorithm 1 Patching Method Solving MVOE Problem

```
Input: T - The element mapping table
Input: A - FPGA architecture information
Input: C - Pre-defined constraints
Output: P - Table of patched information
1   P ← Ø
2   set of elements E ← extract_elements(T)
3   for each element e ∈ E do
4       set of uncertain patterns U ← judge_MVOE(e, T)
5       if U = Ø then
6           Continue
7       else
8           set of neighbors N ← extract_neighbors(e, A)
9           for each pattern u ∈ U do
10              required patterns R ← solve_MVOE(u, N, C)
11              extend P by adding a line including e, u and R
12          end
13      end
14  end
```

Algorithm 1 displays the flow of our patching method for solving the MVOE problem. It requires three inputs, i.e., the element mapping table (**T**), FPGA architecture information (**A**) and a series of pre-defined constraints (**C**), to generate the table of patched information (**P**). Specifically, we first initialize **P** as an empty set Ø (line 1). Then, we extract the configurable elements in the baseline tiles (**E**) from the element mapping table (**T**) as our targets (line 2). Line 3-14 go through **E** to patch information for the special elements. We obtain the uncertain patterns of e (**U**) based on **T** at first (line 4). If **U** is an empty set Ø (line 5), we execute the **Continue** command to analyze the next element (line 6). Otherwise, we abstract all neighbor elements of e (**N**) from **A** (line 8). Then, for each uncertain pattern u, we apply multiple pre-defined constraints (**C**) on e and its neighbors to figure out the required patterns (**R**) of elements in **N** (line 10), and finally, we add e, u, **R** to **P** (line 11). In this way, we can generate a table **P** containing the patches for all MVOE elements in **E**.

Step 4 (Element Reversal): We justify the working pattern for all functional elements based on the bitstream file, the database and our patch information, and then the function of each element can be determined. Now, we can obtain all information about the elements being used as well as the routing paths among them.

Step 5 (Netlist Output): We assemble the routing information and the functional elements into a netlist file.

3) NETLIST REVERSAL TOOL

NRT (Netlist Reversal Tool) retrieves RTL code from the recovered netlist file. The working flow of NRT is shown in Fig. 3c.

Step 1 (Initialization): We read the netlist file from BRT and transform its information, such as element function and routing paths, into a predefined data structure. The FPGA architecture information is also loaded to support code reversing operation.

Step 2 (I/O Determination): We find out the IO ports to generate the ports declaration part in our reversed RTL code. To achieve this goal, we classify all PAD elements in the netlist into two modes, input and output, according to their inherent property. If a PAD works as the source element in a net, it is defined as an input port. Otherwise, it is an output port.

Step 3 (Cluster Extraction): We extract all signal path clusters from the netlist file. A signal path cluster contains multiple signal paths, which are originated from different input PADs, *one PAD for one path* and end with the same output PAD. In this paper, we first track all signal paths belonging to one cluster from an output PAD by using the DFS (Depth-First Search) algorithm [28], and then repeat the above operations until all clusters in the given netlist are found out. Note that one input PAD may be included in multiple signal path clusters. For example, as shown in Fig. 6, two clusters make use of the input PAD2 as their input ports, leading to duplicated port definition during code generation which will be solved in the last step.

Step 4 (Code Generation): We generate RTL code for each cluster. In detail, we separately replace the functional elements with their equivalent RTL code, and combine them together according to the cluster structure and the path properties. For example, the FF (Flip-Flop) in Fig. 6 has two types of signals, data and clock. For the clock signal, we put the clock name "PAD3" into the code "always@()" as the sensitive parameter. For the data signal, we use the data name "PAD2" and "PAD5" to replace the input and output of FF characteristic equation, respectively.

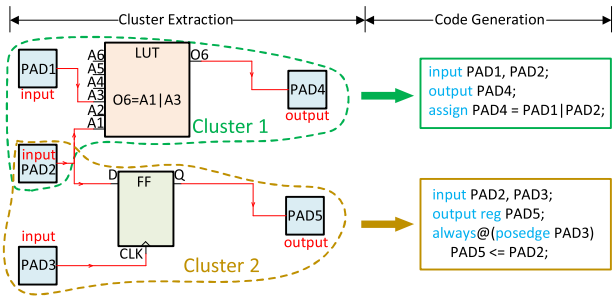


FIGURE 6. An example of cluster extraction and code generation.

Step 5 (RTL Output): We input all RTL code into one file, detect and remove the duplicated input ports, if any, and declare the module name for the whole code.

C. DISCUSSION

We report the time complexity of our tool-chain for each step in Table 1. As we can see, the time complexity of library generator is $O(n + k)$, depending on two variables, i.e., the number of configurable elements in a tile (n) and the number of tiles in the FPGA (k). On the other hand, the time complexity of BRT and NRT are respectively $O(m^2 + n)$ and $O(m^2)$, mainly depending on the number of used components m in the design. In addition, we note that the mapping database constructed by the library generator is only related to the detailed FPGA architecture. Hence, for a given FPGA, we only need to invoke the library generator one time. Subsequently, the constructed mapping database can be re-used for re-engineering any application on that FPGA.

TABLE 1. Time complexity of each step in our tool-chain.

Steps	Time Complexity		
	Library Generator	BRT	NRT
Step 1	*Initialization* O(1)	*Initialization* O(1)	*Initialization* O(1)
Step 2	*Tile Testing* O(n)	*Routing Reversal* O(m^2)	*I/O Determination* O(m)
Step 3	*Offset Computing* O(1)	*Information Patching* O(n)	*Cluster Extraction* O(m^2)
Step 4	*Offset Traversal* O(k)	*Element Reversal* O(m)	*Code Generation* O(m)
Step 5	*Stored Procedure* O(1)	*Netlist Output* O(m)	*RTL Output* O(m)

n is the number of configurable components in a tile;
k is the number of tiles in the FPGA;
m is the number of used components in the design.

IV. EVALUATION

We evaluate our tool-chain both qualitatively and quantitatively.

A. QUALITATIVE ANALYSIS

We compare our work with five existing FPGA reversing tools, namely, Debit [19], BIL [20], BRET [21], Bit2ncd [22] and DAT [25]. Table 2 lists the detailed database and functions in each tool, including FPGA architecture information, bitstream mapping information, bitstream reversing and netlist reversing.

From Table 2, we can observe that our tool-chain is comprehensive for FPGA reverse engineering since it contains all necessary information, i.e., FPGA architecture and bitstream mapping information, as well as functions. In other words, our tool-chain can decompile the intercepted bitstream to RTL code without any support from other tools and databases. In contrast, the existing five tools can only accomplish a part of FPGA reverse engineering since they concern about different phases in re-engineering. Particularly, Debit, BIL, BRET and Bit2ncd put their efforts on bitstream re-engineering while DAT is dedicated to netlist re-engineering. In addition, the four existing bitstream reversing tools may generate a netlist with few errors when targeting complex designs, as their databases and functions remain to be improved, as mentioned in [19]–[22].

TABLE 2. Comparison of FPGA reverse engineering tools.

Works	Database		Function	
	FPGA Arch.	Bitstream Mapping	Bitstream Reversing	Netlist Reversing
Debit [19]	×	◐	◐	×
BIL [20]	●	◐	◐	×
BRET [21]	●	◐	◐	×
Bit2ncd [22]	●	◐	◐	×
DAT [25]	●	×	×	●
Ours	●	●	●	●

●: completely included or implemented, ◐: partly included (sometimes inaccurate) or implemented, ×: not included or implemented.

B. QUANTITATIVE ANALYSIS

To evaluate the effectiveness of our tool-chain, we perform experiments on two well-recognized benchmarks, **ISCAS'85** and **ISCAS'89**, and three real applications, **8051 core**, **68HC08** and **AES**. Xilinx Spartan-6 is taken as our target FPGA and ISE is used to compile the benchmarks to bitstream. Then, we intercept the bitstream and feed it into our tool-chain. In the following experiments, our tool-chain runs on a PC with Intel Core i3-3240 CPU, which works at 3.40GHz main frequency, 4.0 GB RAM, and Windows 7 SP1 32-bit operating system.

1) BITSTREAM REVERSE ENGINEERING

We report the logic cost and bitstream reverse engineering results in Table 3. The logic cost stands for the complexity of our benchmarks, which is quantified by the numbers of FE (Functional Elements in logic tiles and IO tiles) and PS (Programmable Switches). The experimental results evaluate the accuracy and time overhead of our BRT tool.

Our BRT can perfectly decompile a bitstream to netlist as the recovered netlists are totally the same as the original files in both FE and PS for all benchmarks. Moreover, based on the experimental results, we have enough confidence to claim that our tool-chain can achieve 100% reversing correct rate for any design because the 13 benchmarks in Table 3 involve all kinds of components in the target FPGA. We attribute this interesting result to our powerful library generator and patching technique in BRT. As illustrated in Section III, they help to abstract precise bitstream mapping information and solve the MVOE problem, respectively, and therefore, ensuring the accuracy of bitstream reversing results.

Besides, we can observe that the reversing time ranges from 5.23 to 623.21 minutes ($<$ 11 hours), being proportional to the logic complexity of experimental circuits. In general, the time overhead is tolerable for these cases,

TABLE 3. The logic cost and bitstream reverse engineering results for benchmarks.

Benchmark		Logic Cost		Experimental results		
		$FE\ num.$	$PS\ num.$	$FE\ prec.(\%)$	$PS\ prec.(\%)$	$Time(minute)$
ISCAS'85 (Combinational Circuits)	c17	160	55	100.0	100.0	5.23
	c499	2071	2029	100.0	100.0	23.76
	c1908	1822	1733	100.0	100.0	18.01
	c3540	3245	5168	100.0	100.0	40.30
	c6288	7619	10735	100.0	100.0	90.48
ISCAS'89 (Sequential Circuits)	s27	260	269	100.0	100.0	5.75
	s832	1276	1671	100.0	100.0	16.68
	s1423	2855	4012	100.0	100.0	32.58
	s13207	4478	5621	100.0	100.0	48.13
	s38584	22675	35256	100.0	100.0	268.30
Real Applications	8051	29719	45963	100.0	100.0	418.31
	68HC08	30141	48015	100.0	100.0	483.28
	AES	35611	53021	100.0	100.0	623.21

and if necessary, we can shorten the reversing time by applying parallel computing techniques to our tool-chain, e.g., executing the reversing process with multiple parallel threads in GPU (Graphics Processing Unit), which is a special processor widely used in computers for parallel computing [29], [30].

TABLE 4. Netlist reverse engineering results with our tool.

Benchmarks		Time(ms)	Verification
ISCAS'85 (Combinational Circuits)	c17	204	✓
	c499	297	✓
	c1908	279	✓
	c3540	347	✓
	c6288	510	✓
ISCAS'89 (Sequential Circuits)	s27	218	✓
	s832	337	✓
	s1423	327	✓
	s13207	418	✓
	s38584	1386	✓
Real Applications	8051	2073	✓
	68HC08	2455	✓
	AES	3519	✓

2) NETLIST REVERSE ENGINEERING

We decompile the netlist from BRT, report the time cost and verification results in Table 4, and illustrate the correctness of our NRT tool. The time cost is counted from the time when our NRT loads a netlist file to the time when the corresponding RTL code is generated. The formal verification is performed by Synopsys Formality tool, which can detect the difference between the original code and the recovered code in this paper. From Table 4, we can draw the following two conclusions. First, our NRT is a real-time tool since it needs only hundreds to thousands of milliseconds to finish the netlist reverse engineering process for all benchmarks. Second, our NRT tool can precisely transform the netlist to RTL code since all the generated code successfully passes the formal verification. In other words, they are functionally equivalent to the original codes.

3) QUANTITATIVE COMPARISON WITH OTHER WORKS

We compare our tool-chain with existing reversing works, two bitstream reversing tools (Debit [19], Bit2ncd [22]) and one netlist reversing tool (DAT [25]), in reversing accuracy and time cost. To make fair comparison, pong circuit which can control the paddles in an electronic pong game is implemented on a Xilinx Spartan-3 FPGA, as advised in [22].

TABLE 5. Reversing results comparison among works.

Index		Debit	Bit2ncd	DAT	Our Work
Bitstream Reversing	FE accuracy (%)	76.25	100.00	×	100.00
	PS accuracy (%)	69.54	99.52	×	100.00
	Time cost (min)	0.26	23.5	×	38.2
Netlist Reversing	Formal verification	×	×	✓	✓
	Time cost (ms)	×	×	321	254

From Table 5, we can observe that although our tool-chain consumes more time for bitstream reversing, it achieves 100% reversing accuracy in both FE and PS, outperforming Debit and Bit2ncd. Note that a 100% correct netlist is crucial for assisting hardware Trojan detection, in that an inaccurate netlist may distort or eliminate the Trojan logic, reducing the HT detection rate significantly. In addition, both our tool-chain and DAT can retrieve RTL code from the netlist within hundreds of milliseconds, and the two retrieved code files can successfully pass through the formal verification. In other words, the two retrieved code files are functionally equivalent to the original design.

In summary, we can claim that our tool-chain is comprehensive and precise for FPGA reverse engineering. Besides, the accurate bitstream reversing provides the advantageous foundation for future HT detection.

V. CASE STUDY: HARDWARE TROJAN DETECTION

Now, we present a case study to demonstrate the effectiveness of our tool-chain in assisting FPGA HT detection methods.

In detail, we discuss i) how COTD [9] finds out HTs from our recovered netlist, and ii) how coverage analysis [10] identifies the malicious code with the assistance of our tool-chain.

A. HARDWARE TROJAN DETECTION WITH RECOVERED NETLIST

To validate that our tool-chain can assist COTD technique, we perform the experiment on a Trust-Hub benchmark, namely, *s38417* [31]. Specifically, we implement the benchmark on Xilinx Spartan-6 FPGA, intercept the bitstream upon FPGA boot-up and obtain a recovered netlist by using our tool-chain. After that, we feed the recovered netlist into COTD for HT detection.

1) DETECTION IN NETLIST RECOVERED FROM INTACT BITSTREAM

The COTD technique can calculate the controllability and observability of each signal in the netlist, and differentiate the suspicious (potentially malicious) signals from the trusted ones using unsupervised clustering method accordingly. The results of applying COTD on the recovered netlist are reported in Fig. 7. From Fig. 7a, we can observe that COTD splits the signals of recovered, Trojan-infected netlist into two groups. One corresponds to the trustable logics (black dots) and the other refers to Trojan logics (red dots). In contrast, the recovered, Trojan-free netlist contains only trustable logic circuits, as shown in Fig. 7b. Hence, our tool-chain can work as an assistant for COTD technique, i.e., providing recovered netlist, to detect HTs in FPGAs.

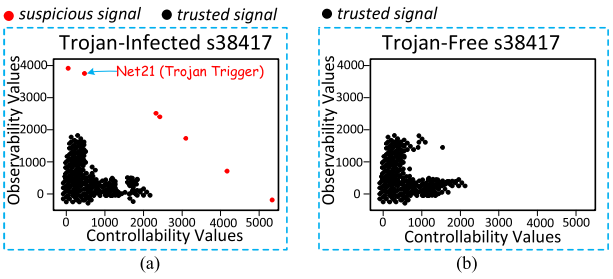
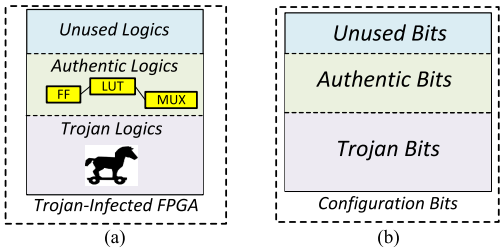


FIGURE 7. Results of COTD analysis in Trojan-infected and -free netlists. (a) Results of COTD analysis in Trojan-infected s38417 netlist. (b) Results of COTD analysis in Trojan-free s38417 netlist.

2) DETECTION IN NETLIST RECOVERED FROM CORRUPTED BITSTREAM

According to Section V-A1, the COTD technique can accurately detect HTs from our recovered netlist. Nevertheless, if the intercepted bitstream gets corrupted (the values of some bits are reversed) due to the poor contact between the probe and configuration lines, the recovered netlist may deviate from the original one and thus affects the HT detection.

To evaluate the influence of bitstream corruptions, we perform a comparative experiment on the Trojan-infected *s38417* benchmark. We first divide the FPGA with Trojans into three parts, i.e., unused logics, authentic logics (used) and Trojan logics (used), as depicted in Fig. 8a. The unused



Corrupted Bits	Reversing Accuracy		Detection Rate (%)
	FE accuracy (%)	PS accuracy (%)	
Unused Bits	100.0	100.0	100.0
Authentic Bits	80.2	73.8	60.0
Trojan Bits	83.4	79.6	20.0

(c)

FIGURE 8. The illustration and results of HT detection in the netlists recovered from corrupted bitstreams. (a) Diagram of experimental FPGA. (b) Diagram of configuration bits in experimental bitstream. (c) Experimental results of reverse engineering and HT detection.

logics are the powered off region in an FPGA, and in contrast, the used logics refer to the activated components. Furthermore, if an active component serves as a part of Trojan circuit, it is contributed to Trojan logics, otherwise, it is authentic logic. Accordingly, as illustrated in Fig. 8b, all configuration bits in a bitstream can also be categorized to three types, corresponding to unused, authentic (used) and Trojan (used) logics. Now, we generate 30 corrupted bitstreams, ten per type, by randomly reversing the value of some bits (in this paper, we assume that 0.1 % configuration bits in a bitstream are corrupted). Afterwards, we recover netlists from these corrupted bitstreams by using our tool-chain and employ the COTD technique to detect HTs in them. The average reversing accuracy and HT detection rate are reported in Fig. 8c.

As shown in Fig. 8c, when the corruption operations occur in unused bits, we can still obtain correct netlists and identify HT in them with 100% detection rate. It is reasonable because the corruptions in the unused bits usually turn on some isolated programmable components which will be marked as abnormal points and removed during the bitstream reversing. As such, they will not introduce any modification to the retrieved netlist. In contrast, when corruptions happen in the authentic or Trojan bits, they will cause some errors on the retrieved netlist, degrading both the re-engineering accuracy and the HT detection rate. In detail, for authentic (Trojan) bits, the average reversing accuracy of Functional Elements and Programmable Switches decreases to 80.2% (83.4%) and 73.8% (79.6%), respectively. In addition, the corruption on Trojan bits results in a worse Trojan detection rate, i.e., 20%, than that of authentic bits (60%).

B. HARDWARE TROJAN DETECTION WITH RECOVERED RTL CODE

We employ our tool-chain to retrieve RTL code from the recovered, Trojan-infected *s38417* netlist, and then utilize the coverage analysis technique to search for malicious logics

in the retrieved code. A well-recognized commercial tool, Synopsys VCS, is used to report the code coverage result automatically and the uncovered lines of code are marked as Trojan [10].

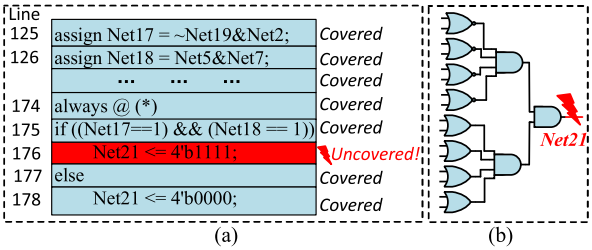


FIGURE 9. The equivalent malicious logic in netlist and RTL code. (a) Uncovered line in the decompiled code. (b) Trojan trigger.

Fig. 9 displays a fraction of Trojan detection results for Trojan-infected *s38417*. As we can see, there is an uncovered code line (line 176 in Fig. 9a) which cannot be executed during our code coverage analysis process. It corresponds to the *Trojan Trigger* signal of Trojan circuit, Net21 in Fig. 9b, which is in accordance with the netlist detection result, as depicted in Fig. 7a. Therefore, our tool-chain can work in conjunction with the coverage analysis technique to identify hardware Trojans.

VI. CONCLUSION

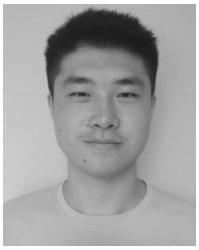
In this paper, we propose a new comprehensive tool-chain which can precisely reverse the FPGA bitstream to RTL code. First, we build an integrated database by conducting thorough black-box testing for configurable FPGA elements and connections. Then, we present two reversing tools, BRT and NRT. With the bitstream mapping and FPGA architecture information in the database, BRT transforms the intercepted bitstream to netlist and NRT further decompiles the netlist to RTL code. After that, we perform extensive experiments to evaluate the effectiveness of our tool-chain. The experimental results reveal that our tool-chain can solely accomplish the whole reversing process from FPGA bitstream to RTL code, and the recovered code is functionally equivalent to the original one. Finally, we present a case study to demonstrate that our tool-chain can assist existing techniques to identify HTs at netlist and RTL code level. From the perspective of hardware security, we believe that our tool-chain can make a significant contribution to thwart HT attacks on FPGAs.

Our tool-chain can only decompile the unencrypted FPGA bitstreams. But this is not a serious problem for some application areas with strict power restriction. For example, terminals in IoT space are preferred to use low-end FPGAs, most of which do not offer bitstream encryption and can be covered by our tool-chain. Also, we note that there are some advanced techniques can decrypt the encrypted FPGA bitstreams, e.g., side channel attack [26], [27], [32], showing a promising direction to extend the applicable scope of our tool-chain.

REFERENCES

[1] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware Trojans: Lessons learned after one decade of research," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 1, pp. 6:1–6:23, 2016.
[2] R. S. Chakraborty, I. Saha, A. Palchaudhuri, and G. K. Naik, "Hardware trojan insertion by direct modification of FPGA configuration bitstream," *IEEE Design Test*, vol. 30, no. 2, pp. 45–54, Apr. 2013.
[3] N. Fern, S. Kulkarni, and K.-T. Cheng, "Hardware Trojans hidden in RTL don't cares—Automated insertion and prevention methodologies," in *Proc. IEEE Int. Test Conf. (ITC)*, Oct. 2015, pp. 1–8.
[4] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan, "Hardware Trojan attacks: Threat analysis and countermeasures," *Proc. IEEE*, vol. 102, no. 8, pp. 1229–1247, Aug. 2014.
[5] T. Iwase, Y. Nozaki, M. Yoshikawa, and T. Kumaki, "Detection technique for hardware Trojans using machine learning in frequency domain," in *Proc. IEEE 4th Global Conf. Consum. Electron. (GCCE)*, Oct. 2015, pp. 185–186.
[6] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith, "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," in *Proc. IEEE Symp. Secur. Privacy*, May 2010, pp. 159–172.
[7] X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, and P. Mishra, "Pre-silicon security verification and validation: A formal perspective," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, 2015, pp. 1–6.
[8] J. Zhang, F. Yuan, L. Wei, Y. Liu, and Q. Xu, "VeriTrust: Verification for hardware trust," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 7, pp. 1148–1161, Jul. 2015.
[9] H. Salmani, "COTD: Reference-free hardware trojan detection and recovery based on controllability and observability in gate-level netlist," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 2, pp. 338–350, Feb. 2017.
[10] A. Nahiyan and M. Tehranipoor, "Code coverage analysis for IP trust verification," in *Hardware IP Security and Trust*. Cham, Switzerland: Springer, 2017, pp. 53–72.
[11] S. E. Quadir *et al.*, "A survey on chip to system reverse engineering," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 1, p. 6, 2016.
[12] R. Courtland, "3D X-ray tech for easy reverse engineering of ICs," *IEEE Spectr.*, vol. 54, no. 5, pp. 11–12, May 2017.
[13] R. Torrance and D. James, "The state-of-the-art in IC reverse engineering," in *Proc. 11th Int. Workshop Cryptograph. Hardw. Embedded Syst.*, 2009, pp. 363–381.
[14] W. Li *et al.*, "WordRev: Finding word-level structures in a sea of bit-level gates," in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, Jun. 2013, pp. 67–74.
[15] T. Meade, S. Zhang, M. Tehranipoor, and Y. Jin, "A comprehensive netlist reverse engineering toolset for IC trust," in *Proc. Government Microcircuit Appl. Crit. Technol. Conf.*, 2016, pp. 281–284.
[16] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, "Reverse engineering digital circuits using functional analysis," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2013, pp. 1277–1280.
[17] P. Subramanyan *et al.*, "Reverse engineering digital circuits using structural and functional analyses," *IEEE Trans. Emerg. Topics Comput.*, vol. 2, no. 1, pp. 63–80, Mar. 2014.
[18] D. Ziener, S. Assmus, and J. Teich, "Identifying FPGA IP-cores based on lookup table content analysis," in *Proc. Int. Conf. Field Program. Log. Appl.*, Aug. 2006, pp. 1–6.
[19] J. Note and E. Rannaud, "From the bitstream to the netlist," in *Proc. 16th Int. ACM/SIGDA Symp. Field Program. Gate Arrays*, 2008, p. 264.
[20] F. Benz, A. Seffrin, and S. A. Huss, "Bil: A tool-chain for bitstream reverse-engineering," in *Proc. 22nd Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2012, pp. 735–738.
[21] J. Yoon *et al.*, "A bitstream reverse engineering tool for FPGA hardware trojan detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 2318–2320.
[22] Z. Ding, Q. Wu, Y. Zhang, and L. Zhu, "Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation," *Microprocessors Microsyst.*, vol. 37, no. 3, pp. 299–312, 2013.
[23] C. Wolf. (2015). *Project IceStorm*. [Online]. Available: http://www.clifford.at/icestorm/
[24] P. Swierczynski, G. T. Becker, A. Moradi, and C. Paar, "Bitstream fault injections (BiFI)–automated fault attacks against SRAM-based FPGAs," *IEEE Trans. Comput.*, vol. 67, no. 3, pp. 348–360, Mar. 2018.

[25] D. I. Cheremisinov, "Design automation tool to generate EDIF and VHDL descriptions of circuit by extraction of FPGA configuration," in *Proc. East-West Design Test Symp. (EWDTS)*, Sep. 2013, pp. 1–4.
[26] A. Moradi, A. Barenghi, T. Kasper, and C. Paar, "On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from Xilinx Virtex-II FPGAs," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 111–124.
[27] A. Moradi, D. Oswald, C. Paar, and P. Swierczynski, "Side-channel attacks on the bitstream encryption mechanism of Altera Stratix II: Facilitating black-box analysis using software reverse-engineering," in *Proc. ACM/SIGDA Int. Symp. FPGA*, 2013, pp. 91–100.
[28] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, pp. 558–562, 1962.
[29] S. Devadithya, A. Pedross-Engel, C. M. Watts, N. I. Landy, T. Driscoll, and M. S. Reynolds, "GPU-accelerated enhanced resolution 3-D SAR imaging with dynamic metamaterial antennas," *IEEE Trans. Microw. Theory Techn.*, vol. 65, no. 12, pp. 5096–5103, Dec. 2017.
[30] G. Zhou, R. Bo, L. Chien, X. Zhang, S. Yang, and D. Su, "GPU-accelerated algorithm for online probabilistic power flow," *IEEE Trans. Power Syst.*, vol. 33, no. 1, pp. 1132–1135, Jan. 2018.
[31] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, Oct. 2013, pp. 471–474.
[32] A. Moradi and T. Schneider, "Improved side-channel analysis attacks on Xilinx bitstream encryption of 5, 6, and 7 series," in *Proc. Int. Workshop Constructive Side-Channel Anal. Secure Design*, 2016, pp. 71–87.



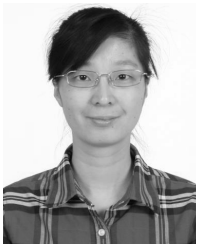
TAO ZHANG received the B.S. degree in communication engineering from Northwest University, Xi'an, China, in 2016. He is currently pursuing the M.S. degree with the School of Information and Communication Engineering, University of Electronic Science and Technology of China.

His research focuses on hardware security and trust, with a special interest on hardware reverse engineering.

JIAN WANG (M'16) received the M.S. and Ph.D. degrees from the University of Electronic Science and Technology of China, Chengdu, China, in 2008 and 2011, respectively, where he has been an Associate Professor, since 2014.

His current research interests include hardware security and on-chip communication architectures, with a special interest on hardware Trojan detection and hardware reverse engineering.

SHIZE GUO received the B.S. and M.S. degrees from the Ordnance Engineering College, China, in 1988 and 1991, respectively, and the Ph.D. degree from the Harbin Institute of Technology, in 1994. He is currently a Professor with the DSP Laboratory, University of Electronic Science and Technology of China, Chengdu, China. His main research interests include information technology and information security, with a special interest on hardware security and trust.



ZHE CHEN received the M.S. and Ph.D. degrees from the Beijing University of Posts and Telecommunications, in 2005 and 2008, respectively.

She is currently an Associate Professor with the DSP Laboratory, University of Electronic Science and Technology of China, Chengdu, China. She has published more than 50 technical papers in peer-reviewed journals and conferences. Her research interest includes hardware security, especially the on-chip systems design and test.