Graph Similarity and its Applications to Hardware Security

Marc Fyrbiak, Sebastian Wallat, Sascha Reinhard, Nicolai Bissantz, and Christof Paar, *Fellow, IEEE*

Abstract—Hardware reverse engineering is a powerful and universal tool for both security engineers and adversaries. From a defensive perspective, it allows for detection of intellectual property infringements and hardware Trojans, while it simultaneously can be used for product piracy and malicious circuit manipulations. From a designer's perspective, it is crucial to have an estimate of the costs associated with reverse engineering, yet little is known about this, especially when dealing with obfuscated hardware. The contribution at hand provides new insights into this problem, based on algorithms with sound mathematical underpinnings. Our contributions are threefold: First, we present the graph similarity problem for automating hardware reverse engineering. To this end, we improve several state-of-the-art graph similarity heuristics with optimizations tailored to the hardware context. Second, we propose a novel algorithm based on multiresolutional spectral analysis of adjacency matrices. Third, in three extensively evaluated case studies, namely (1) gate-level netlist reverse engineering, (2) hardware Trojan detection, and (3) assessment of hardware obfuscation, we demonstrate the practical nature of graph similarity algorithms.

Index Terms—Graph similarity, hardware reverse engineering, hardware trojan, hardware obfuscation assessment

1 Introduction

In times of globalized Integrated Circuit (IC) design and offshore fabrication processes, the need for protection of valuable Intellectual Property (IP) assets and detection of manipulations such as hardware Trojans has highly increased [1]. To mitigate these serious risks for Application Specific Integrated Circuits (ASICs) as well as Field Programmable Gate Arrays (FPGAs), security engineers are forced to resort to reverse engineering to witness IP infringement in competitors' products [2] or to detect malicious design manipulations [3] (e.g., in untrusted third-party IP cores), since the Register Transfer Level (RTL) source code is typically not available in these scenarios. Unfortunately, manual reverse engineering is a time-consuming task even for an experienced team of analysts, thus automated and reliable techniques are inevitable to reduce time and costs.

In addition to being useful for actual reverse engineering applications, understanding of automated reverse engineering also provides valuable guidelines for threat estimation against (powerful) attackers, and it aids with the design of sound protective countermeasures such as hardware obfuscation or physical design obfuscation [4]. For example, numerous solutions have been proposed to protect IP against illegitimate reuse or modification, see Shakaya et al. [5] for a comprehensive overview. Although many of these schemes seem promising, they typically neglect automated reverse engineering capabilities to analyze hardware designs armed with obfuscation features (e.g., see Wallat et al. [6]). Here, insights in reverse engineering facilitates improved countermeasures to mitigate aforementioned risks.

In typical real-world scenarios (e.g., detection of hardware Trojans in third-party IP cores or analysis of a competitor product), a reverse engineer has access to a flattened, unstructured gate-level netlist of the target design. From a high-level point of view, there are two technical key challenges for automated gate-level netlist reverse engineering: (1) module boundary and hierarchy recovery, and (2) matching to known library components. To infer high-level functionality of an unstructured netlist, an analyst has to identify boundaries of *candidate modules* to subsequently analyze their hierarchy, cf. Subramanyan et al. [7]. Typically, extracted candidate modules are matched to a set of library modules (e.g., counters or cryptographic Sboxes) with Boolean function analysis or subgraph isomorphism. However, these approaches suffer from limitations of reliability in case of imperfect netlist recovery or design obfuscation. Here, even minor errors typically lead to unrecognized candidate modules since both matchings techniques require strict information. For example, chip-level reverse engineering uses imperfect image processing [2], which is prone to faults such as incorrectly assigned or missing signals, or incorrectly recovered gate types. In addition, design obfuscation, different optimization strategies, and diversity of implementation strategies challenges identification of candidate modules even if an error-free netlist is available.

Goals and Contributions. In this work, we focus on detection of similarities between gate-level netlists rather than exact matchings of Boolean function analysis or subgraph isomorphism. Our goal is to examine its suitability for the hardware security domain. This approach seems promising

- M. Fyrbiak, S. Reinhard, and N. Bissantz are with the Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, 44801 Bochum, Germany. E-mail: {marc.fyrbiak, sascha.reinhard, nicolai.bissantz}@rub.de.
- S. Wallat is with the University of Massachusetts Amherst, Amherst, MA 01003. E-mail: swallat@umass.edu.
- C. Paar is with the Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, 44801 Bochum, Germany, and also with the University of Massachusetts Amherst, Amherst, MA 01003. E-mail: christof.paar@rub.de.

Manuscript received 10 Sept. 2019; accepted 11 Oct. 2019. Date of publication 26 Nov. 2019; date of current version 10 Mar. 2020.
(Corresponding author: Sebastian Wallat.)
Recommended for acceptance by T. Li.
Digital Object Identifier no. 10.1109/TC.2019.2953752

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

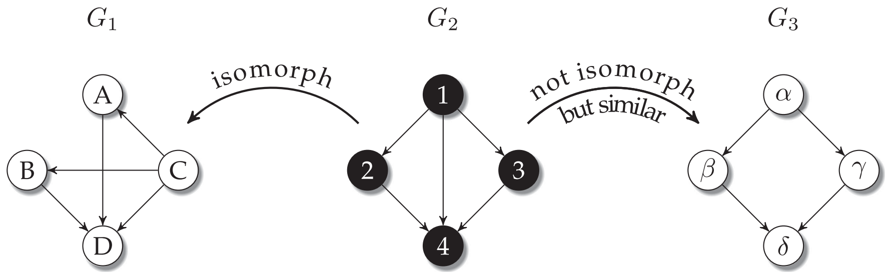


Fig. 1. Example of the difference between isomorphism and similarity. G_1 and G_2 are isomorph ($f(1) = C,\ f(2) = A,\ f(3) = B,\ f(4) = D$), whereas G_2 and G_3 are not isomorph, even though they are topologically similar. The missing edge for an isomorphism from G_2 to G_3 is (α, δ).

since these heuristics have been used successfully in several other settings, including malware detection [8], [9], grading of programming assignments [10], bioinformatics and data mining [11]. To this end, we first analyze characteristics of hardware netlists to improve state-of-the-art similarity heuristics through tailored optimizations. Subsequently, we introduce our novel approach based on spectral analysis of adjacency matrices. Finally, in three case studies, we demonstrate the efficacy of similarity analysis for large and complex hardware designs. In summary, our main contributions are:

- *Graph Similarity for Hardware Security.* To the best of our knowledge, we are the first to apply the graph similarity problem in the hardware security domain. We show a broad spectrum of graph similarity applications by means of three case studies, namely (1) gate-level reverse engineering of security-relevant circuitry, (2) detection of hardware Trojans, and (3) assessment of hardware obfuscation. To this end, we improve state-of-the-art similarity heuristics in terms of accuracy and computation time by optimizations and novel preprocessing techniques tailored to the hardware setting.
- *Novel Similarity Heuristic.* We present a novel graph similarity heuristic based on spectral graph analysis. More precisely, we analyze spectral information of of two graphs in a multiresolutional way to determine their similarity. Eigenvalues of the graph's adjacency matrices are computed and a suitable distance measure between respective eigenvalue distributions is determined.
- *Extensive Evaluation.* Our evaluation demonstrates the efficacy of graph similarity heuristics for large hardware benchmarks while keeping analysis time practical. Additionally to the variety of algorithms, our evaluation covers different Field Programmable Gate Array (FPGA) families and several design optimization goals to emphasize the reliability of our approach for each case study.

2 System Model

We assume a reverse engineer with access to a 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 reverse engineer is able to be able to retrieve information of the design's internal workings for a specific purpose (e.g., hardware Trojan detection, competitor analysis, or finding evidence of IP infringement). The gate-level netlist can be obtained through several means: (1) chip-level or layout reverse engineering [4] in case of ASICs, (2) bitstream-level reverse engineering [12] in case of FPGAs, (3) directly from an IP provider [1].

Note that this system model is in line with prior research on hardware security [3].

3 The Graph Similarity Problem

Before we detail a variety of state-of-the-art graph similarity heuristics and our hardware-specific improvements, we provide essential background on graph similarity and the notation used throughout the remainder of this work. Note that we represent gate-level netlists as graphs and leverage similarity algorithms between graphs for hardware reverse engineering.

3.1 Preliminaries

Definition 1 (Directed Graph). *A digraph $G = (V, E)$ is a pair where V is a set of vertices, and $E \subseteq V \times V$ is a set of edges (ordered pairs of vertices). $d^-_G(v)$ denotes outgoing edges for a vertex v in G, and $d^+_G(v)$ denotes ingoing edges, respectively. $c_G(v)$ is the set of child vertices for a vertex v in G, i.e., the projection $c_G(v) = \pi_1(d^-_G(v)) := \{\pi_1(v, v_a), \ldots, \pi_1(v, v_z)\} = \{v_a, \ldots, v_z\}$. $p_G(v)$ is the set of parent vertices for a vertex v in G, i.e., the projection $\pi_0(d^+_G(v))$. The function $\mathsf{label} : V \to \mathbb{N}$ determines the label of a vertex.*

Two relationships are relevant for our work: (1) isomorphism, and (2) similarity. From a high-level perspective, graph isomorphism captures whether two graphs are structurally equivalent or not. Graph similarity relaxes this binary decision to a real number indicating a level of similarity, see Fig. 1.

Definition 2 (Graph Isomorphism). *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. G_1 and G_2 are isomorph, if there exists a bijection $f : V_1 \to V_2$ such that $\forall (u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$.*

Definition 3 (Graph Similarity Algorithm). *Let G_1 and G_2 be two graphs. A graph similarity algorithm $\mathcal{A} : (G_1, G_2) \to [0, 1]$ computes a real-valued similarity score for G_1 and G_2. A similarity score of 1 indicates that G_1 and G_2 are identical.*

In order to effectively measure similarity, we use the notion of graph edit distance, see Definition 4. The edit distance measures the smallest number of edit operations transforming one graph into another one.

Definition 4 (Graph Edit Distance). *Let G_1 and G_2 be two graphs. The graph edit distance is a function $\mathsf{GED}(G_1, G_2) \rightarrow \mathbb{N}$ which computes the smallest number of edit operations to transform G_1 into G_2. The edit operations are: adding an isolated vertex e_v^+, deleting an isolated (without connecting edges) vertex e_v^-, adding an edge e_e^+, deleting an edge e_e^-, relabeling a vertex e_v^r. Each edit operation has a specific cost, defined by $\mathsf{cost} : \{e_v^+, e_v^-, e_e^+, e_e^-, e_v^r\} \rightarrow \mathbb{N}$, typically 1 for vertex-edit and edge-edit operations.*

Even though the problem of graph isomorphism and graph edit distance are conceptionally easy to understand, both are hard to solve in a generic way: computation of the graph edit distance is NP-hard (exponential in number of vertices/edges), the graph isomorphism problem is in the low hierarchy of NP, and the subgraph isomorphism problem is NP-complete [13]. Over the years, various heuristic algorithms have been proposed to provide a similarity measure. These heuristics often involve scenario-specific optimizations (1) to increase accuracy, and (2) to reduce computation time via reduction of analyzed graphs.

3.2 Hardware Characteristics and Optimizations

Since we deal with graphs representing hardware (i.e., gate-level netlists), we now describe algorithm-specific optimizations based on general hardware characteristics, namely (1) vertex labeling, and (2) subgraph analysis. Our optimizations increase the accuracy and reliability of graph similarity heuristics and simultaneously enable major reduction of computation times. Note that graph similarity algorithms may have to be adapted to incorporate these optimizations.

Vertex Labeling. Since we target graphs representing gate-level netlists, vertices represent gates that implement Boolean functions. To effectively distinguish vertices, each gate type is assigned a specific label (e.g., an XOR gate is labeled *xor*, an AND gate is labeled *and*, etc.). To be more precise, we use natural numbers to represent labels, cf. Definition 1. Note that typical hardware libraries may contain up to one hundred or more atomic gates. Thus, we may have to adapt similarity algorithms to support labels.

Subgraph Analysis. Since modern hardware designs are typically assembled from a variety of modules such as IP cores, our main focus is to identify these *small* (e.g., 100 vertices) subgraphs in a *large* (e.g., 5,000 vertices) design graph rather than a similarity analysis for two large graphs. Therefore, we may have to adapt similarity algorithms to support a subgraph analysis rather than two equally-sized graphs.

3.3 Graph Similarity Preprocessing Strategies

We now provide a high-level overview of our two-phased graph similarity analysis using different netlist preprocessing steps. From a high-level point of view, *phase 1* represents a fast coarse-grained similarity analysis, however, *phase 1* is inevitably prone to false-positive identified similarities. To overcome its fundamental limitations, we perform a slower but fine-grained and thus more reliable *phase 2*.

3.3.1 Phase 1: Combinational Logic Subgraphs

A typical hardware design consists of combinational logic implementing Boolean functions to transform data stored in Flip Flops (FFs) forming registers. In particular, combinational logic between register stages are interesting for the human analyst since they implement crucial Boolean functions (e.g., a hardware Trojan trigger). Therefore, we analyze graph similarity among combinational logic subgraphs rather than the whole graph. This approach yields both increased accuracy since registers are a potential pitfall for false-positives and reduced computation time since combinational logic subgraphs are significantly smaller and can be analyzed in parallel.

To determine combinational logic subgraphs, we process the design in a two-phased approach. First, we determine so-called *register groups* [14]. To be more precise, we group all FFs which have equal control signals (e.g., clock, chip enable, or (a)synchronous (re)set). Second, for each Flip Flop (FF) in each register group we perform a reverse breath-first search until we reach a FF. Here, *reverse* means that we change direction of each edge. Each combinational logic gate visited during reverse breath-first search is added to the combinational logic group for the register group. Note that we also report the register group size, since this information yields valuable information about the design's architecture for the human analyst. For example, the register grouping identifies general-purpose registers of a Central Processing Unit (CPU) or the datapath of a crypto implementation, see Section 4.

In summary, *phase 1* analyzes similarities among combinational logic subgraphs of both hardware designs.

3.3.2 Phase 2: Combinational Logic Bitslices and LUT Decomposition

Combinational Logic Bitslices. Even though combinational logic subgraphs are significantly smaller than the original graph, they can still consist of numerous gates (e.g., a datapath of a CPU or crypto algorithm). In order to further reduce the size of the subgraphs, we analyze so-called *bitslices* [7] of combinational logic subgraphs. More precisely, a bitslice is a Boolean function with one output and multiple inputs. Hence, each output signal of a combinational logic subgraph yields a single bitslice. Our bitslice analysis is based on the observation that similar combinational logic subgraphs share similar bitslice subgraphs, and analysis of bitslices provides a more fine-grained similarity value. Analogous to our combinational logic subgraph generation, we perform a reverse breath-first search for each output signal until we reach inputs of the subgraph. Each visited gate is added to the bitslice.

For each combinational logic subgraph, we do not obtain one but multiple similarity values since we compare multiple bitslices. Even though a human analyst is capable to analyze such a vector of similarity values, we found it practical to reduce this number to a single values. To this end, we simply determine the arithmetic mean of the similarity values. Note that we also report the standard deviation in case similarity values are spread over a wide range of values.

FPGA LUT Decomposition. We now describe a netlist preprocessing technique tailored to FPGA designs which on one hand significantly increases the accuracy of graph similarity algorithms, but on the other hand increases the size of the analyzed graphs. A crucial building block of FPGAs are so-called Look-up tables (LUTs), typically small truth tables (with 2 to 6 inputs) which implement Boolean functions and thus form combinational logic of a hardware design (along with other dedicated multiplexers or carry gates). From a graph theory perspective, each Look-up table (LUT) is treated as a single vertex regardless of its implemented Boolean function, so even if a LUT L_1 implements a simple Boolean OR and a LUT L_2 implements (parts of) a highly non-linear cryptographic Sbox, both L_1 and L_2 are treated equally even if labeling is used.

To address this fundamental limitation, we preprocess the target gate-level netlist and replace each LUT with its implemented Boolean function. More precisely, we determine the minimal form of a Boolean function with the Quince-McCluskey algorithm [15] in order to represent each LUT with the minimum number of AND-OR-INV logic gates. Note that the Quince-McCluskey algorithm's runtime grows exponentially with the number of variables, however, typical LUTs have a small input size ($\leq$6), hence this is not a limitation in practice. This preprocessing step naturally increases the netlist size, however, this strategy enables to address the aforementioned issue in a generic way, i.e., independent of any graph similarity heuristic. Furthermore, this step unifies netlists of different LUT architectures.

In summary, *phase* 2 analyzes similarities among combinational logic bitslices of both hardware designs whose LUTs have been decomposed.

Similarity Algorithms. In the following, we present two state-of-the graph similarity algorithms (Sections 3.4 and 3.5) including our adaptions in the hardware context. Subsequently, we present our multiresolutional spectral analysis (Section 3.6). In addition to the presented graph similarity algorithms, we evaluated the applicability of maximum common subgraph and VF2 subgraph isomorphism (implemented in Boost). However, either both searches identifies that no subgraph is found in a range of seconds (even using combinational subgraph preprocessing), or its computation time is beyond several hours thus we found both approaches to be impractical for our evaluation. Note that such a mismatch occurs due to multi-level circuit minimization.

Furthermore, we implemented and evaluated the applicability of a label transition systems approach by Sokolsky et al. [16] and the k-subgraph analysis by Kruegel et al. [8]. However, computation time of the labeled transition system is beyond several days for larger graphs and thus impractical for our evaluation. Moreover the k-subgraph analysis provides inaccurate similarity results for our case studies even though computation time is practical (several minutes up to hours for selected hardware designs). Note that we adapted the original subgraph generation by Kruegel et al. to cope with hardware graphs where nodes may have more than 2 successors.

We refer the reader to Section 5 for details on our algorithm selection.

3.4 Graph Edit Distance Approximation

Although the graph edit distance effectively measures similarity of two graphs, its computational complexity is a fundamental drawback. Hu et al. [9] proposed an algorithm to approximate the graph edit distance to measure similarities among software function-call graphs for malware detection. The key idea of this algorithm is to analyze edit operations to map each vertex in the two graphs, and then leverage the Hungarian method that solves this assignment problem[1] in $\mathcal{O}(|V|^3)$ polynomial time [17]. The Hungarian method finds the optimal assignment, i.e., a matching for vertex sets with minimal cost.

Algorithm 1. Graph Edit Distance Approximation

```
Input: Graph $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$,
       Boolean $label$, Boolean $subgraph$
Output: Similarity Score $s \in [0, 1]$ for $G_1$ and $G_2$
    Vertex matching $m \in \mathbb{N}^{(|V_1|+|V_2|)\times(V_1|+|V_2|)}$
    // initialization of cost matrix
 1: matrix $(c_{ij}) \in \mathbb{N}^{(|V_1|+|V_2|)\times(|V_1|+|V_2|)}$ initialized with 0
 2: for vertex $v_i \in V_1$ then
 3:   for vertex $v_j \in V_2$ do
 4:     $c_{ij} \leftarrow$ edit_distance($v_i$, $v_j$, $label$, $subgraph$)
 5: for row $i$ with $0 \leq i < |V_1|$ do
 6:   for column index $j$ with $0 \leq j < |V_1|$ do
 7:     column index $k \leftarrow j + |V_2|$
 8:     if $i = j$ then
 9:       $c_{ik} \leftarrow |d^{+}_{G_1}(v_i)| + |d^{-}_{G_1}(v_i)| + 1$
10:     else
11:       $c_{ik} \leftarrow \infty$
12: if $subgraph = false$then
13:   for row index $i$ with $0 \leq i < |V_2|$ do
14:     for column index $j$ with $0 \leq j < |V_2|$ do
15:       row index $k \leftarrow i + |V_1|$
16:       if $i = j$ then
17:         $c_{kj} \leftarrow |d^{+}_{G_2}(v_i)| + |d^{-}_{G_2}(v_i)| + 1$
18:       else
19:         $c_{kj} \leftarrow \infty$
    //search optimal vertex matching
20: vertex matching $m \leftarrow$ hungarian($(c_{ij})$) $\in \mathbb{N}^{(|V_1|+|V_2|)^2}$
    //similarity computation
21: if $subgraph = false$ then
22:   return $s \leftarrow 1 - \frac{\sum_{i \leftarrow 0}^{|V_1|+|V_2|-1} c_{im_i}}{|V_1|+|V_2|+2(|E_1|+|E_2|)}$
23: else
24:   return $s \leftarrow 1 - \frac{\sum_{i \leftarrow 0}^{|V_1|+|V_2|-1} c_{im_i}}{2|V_1|+4|E_1|}$
```

Optimizations Tailored to Hardware. Hu et al. [9] described an optimization similar to vertex labeling to increase accuracy. In our case, we incorporate vertex labeling described in Section 3.1. Also, more importantly for our case, we add a subgraph search capability to the algorithm to measure similarity for a small subgraph rather than two equally sized graphs. Note that in the subgraph search, we assume that G_1 is the small subgraph and G_2 is the larger one. In case the subgraph search is not used, but G_1 is significantly smaller than G_2 we obtain a small similarity value. Note that we parameterized optimizations to measure its impact on the heuristic's accuracy in our evaluation, see Section 4.

Algorithm 1 shows the graph edit distance approximation extended with our optimizations. First, the quadratic cost matrix (c_{ij}) is initialized (❶-⓳). Note that $|V_2|$ dummy

[1] The assignment is defined as follows: Let S, T be two sets of equal size and let $c : S \times T \rightarrow \mathbb{R}$ be a cost function. The goal is to find a bijection $f : S \rightarrow T$ such that the cost $\sum_{s \in S} c(a, f(a))$ is minimized.

TABLE 1
Edit Cost Equations for Parameter Configurations Label and Subgraph

label	*subgraph*	Equation
false	*false*	$ed \leftarrow \max(\lvert d^+_{G_1}(v_i)\rvert, \lvert d^+_{G_2}(v_j)\rvert) - \min(\lvert d^+_{G_1}(v_i)\rvert, \lvert d^+_{G_2}(v_j)\rvert) + \max(\lvert d^-_{G_1}(v_i)\rvert, \lvert d^-_{G_2}(v_j)\rvert) - \min(\lvert d^-_{G_1}(v_i)\rvert, \lvert d^-_{G_2}(v_j)\rvert)$
true	*false*	$ed \leftarrow \lvert p_{G_1}(v_i)\rvert + \lvert p_{G_2}(v_j)\rvert - 2\lvert \mathsf{C}(p_{G_1}(v_i) \cap p_{G_2}(v_j))\rvert + \lvert c_{G_1}(v_i)\rvert + \lvert c_{G_2}(v_j)\rvert - 2\lvert \mathsf{C}(c_{G_1}(v_i) \cap c_{G_2}(v_j))\rvert$
false	*true*	$ed \leftarrow \lvert d^+_{G_1}(v_i)\rvert - \min(\lvert d^+_{G_1}(v_i)\rvert, \lvert d^+_{G_2}(v_j)\rvert) + \lvert d^-_{G_1}(v_i)\rvert - \min(\lvert d^-_{G_1}(v_i)\rvert, \lvert d^-_{G_2}(v_j)\rvert)$
true	*true*	$ed \leftarrow \lvert p_{G_1}(v_i)\rvert - \lvert \mathsf{C}(p_{G_1}(v_i) \cap p_{G_2}(v_j))\rvert + \lvert c_{G_1}(v_i)\rvert - \lvert \mathsf{C}(c_{G_1}(v_i) \cap c_{G_2}(v_j))\rvert$

vertices are added to V_1, and $|V_1|$ dummy vertices to V_2. The quadratic cost matrix is split into four equally sized parts. The top left part denotes the edit distance to transform a vertex from V_1 into a vertex from V_2 (❷-❹). Note that the function edit_distance is described below. The top right part denotes cost to transform a vertex from V_1 to a dummy vertex from V_2, i.e., deletion of ingoing $|d^-_G(v)|$ and outgoing $|d^+_G(v)|$ edges as well as vertex deletion (❺-⓫). Similarly, the bottom left part denotes cost to transform a vertex from V_2 to a dummy-vertex from V_1 (⓭ -⓳). The bottom right part denotes cost to transform dummy vertices from V_1 into dummy vertices from V_2, which has zero associated cost ❶. Second, the Hungarian method is used to find the optimal vertex matching with smallest costs between vertex sets V_1 and V_2 (initialization in⓴). Third, graph similarity is computed (㉑ -㉔). Note that the original algorithm computed an approximation of the graph edit distance and not a similarity score, Chan et al. [18] defined a formula to compute this value for a given edit distance ㉒. The computational complexity of the graph edit distance approximation is $\mathcal{O}((|\mathcal{V}_1| + |\mathcal{V}_2|)^6)$ since the Hungarian method can be implemented with $\mathcal{O}(|V|^3)$.

In case of a subgraph search (*subgraph* = *true*) we omit the bottom left part⓬, i.e., cost to transform a vertex from V_2 to dummy vertices from V_1. Thus, we can remove these vertices from V_2 at no cost since we are only interested in measuring cost of identifying the small subgraph G_1 in G_2. Furthermore, we adjust the denominator in the similarity computation ㉔ to the highest number of edit operations to transform a subgraph of G_2 in G_1, i.e., delete and add $|V_1|$ vertices (cost of $2|V_1|$) as well as delete and add $|E_1|$ edges (cost of $4|E_1|$). Note that the cost of edge edit operations are 2, see Definition 1.

Edit Distance Computation. To compute the edit$_{\text{d}}$istance ❹, we use distinct strategies for all 4 different parameter possibilities, see Table 1. Equation (1) states the original edit distance cost formula by Chan et al. [18]

$$ed \leftarrow \max(|d^+_{G_1}(v_i)|, |d^+_{G_2}(v_j)|) - \min(|d^+_{G_1}(v_i)|, |d^+_{G_2}(v_j)|) + \max(|d^-_{G_1}(v_i)|, |d^-_{G_2}(v_j)|) - \min(|d^-_{G_1}(v_i)|, |d^-_{G_2}(v_j)|). \tag{1}$$

In case that both parameters *label* and *subgraph* are *false*, Equation (1) computes the edit distance. For vertex labeling optimization $(label = true, subgraph = false)$, we count occurrences of vertex labels in parent and child sets instead of the number of ingoing d^+_G and outgoing d^-_G edges. Formally, the input for the function C in Table 1 is a set of vertices and it returns a multiset of vertex labels with their multiplicity. For subgraph optimization $(label = false, subgraph = true)$, we omit these edit distance costs for G_2 since we are only interested in subgraph G_1. If both parameters are *true*, both strategies are combined.

In our implementation, we also return the vertex matching m to analyze similar vertices in both graphs. To detect multiple matchings, for example if a hardware unit is instantiated multiple times, the algorithm is executed again but we initialize costs for already matched vertices to ∞. Furthermore, we implemented all *for* loops in ❷-⓳ in a parallel fashion to speed up execution.

3.5 Neighbour Matching

In addition to graph edit distance approximation, another strategy to address the issue was proposed by Vujošević-Janišić et al. [10]. The key idea of this algorithm is to analyze the graph topology and match neighboring vertices. To this end, a similarity submatrix is built that compares topology of a vertex, and then the Hungarian method is leverged to solve this assignment problem. Similar to graph edit distance approximation, the Hungarian method finds the optimal assignment for this matrix, i.e., a matching for vertex sets with minimal cost.

Optimizations Tailored To Hardware. Since Vujošević-Janičić et al. [10] developed an algorithm to compare software implementations, their vertex labeling focuses on instructions. In our case, we incorporate the vertex labeling described in Section 3.1 and we adjusted the final similarity score computation by a subgraph search. As noted before, we assume that G_1 is the small subgraph and G_2 is the large one and we parameterize each optimization to measure its impact on the algorithm's accurarcy.

Algorithm 2 shows the neighbour matching extended with our optimizations. First, the topological similarity matrix (sim_{ij}) is initialized (❶ -❼). In case of vertex labeling $(label = true)$, we utilize the experimentally determined value 0.5 to distinguish vertices with different labels, i.e., $(\mathsf{label}(v_i) \neq \mathsf{label}(v_j))$. Note that a value in range $[0.1 - 0.3]$ resulted in higher false negative rate and smaller false positive rate, values in range $[0.7 - 0.9]$ caused a higher false positive rate. Second, the similarity matrix is iteratively updated until each element in the temporary matrix (tmp_{ij}) is not larger than some chosen precision ϵ ❾, i.e., $\epsilon = 10^{-4}$ [18]. In each iteration (❿ -㉘), the new similarity matrix (sim_{ij}) is determined as follows: for vertex pair (v_i, v_j) the optimal vertex matching is computed with the Hungarian method ⓳. Based on this vertex matching, the vertex similarity value is determined for the parent (input) vertices sim_{in} (line ⓴) and the child (output) vertices sim_{out} ㉘. Note in case both vertices have no parents or children, we set the vertex similarity value to 1. Finally, the new similarity value sim_{ij} is determined (㉙ -㉜). Third, after the similarity matrix stabilizes, the similarity value s is computed (㉝ -㊲). Therefore, the Hungarian method is applied again to find the optimal vertex matching. For subgraph search $(sg = true)$, we adjust the denominator since we are only interested to determine a subgraph in G_2 similar to graph G_1.

Algorithm 2. Neighbour Matching

```
Input: Graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$,
       Boolean $label$, Boolean $sg$ (subgraph), $\epsilon > 0$
Output: Similarity score $s \in [0, 1]$ for $G_1$ and $G_2$
    // initialization of the sim. matrix
 1: matrix $(sim_{ij}) \in \mathbb{R}^{|V_1| \times |V_2|}$
 2: for vertex $v_i$ in $V_1$ do
 3:   for vertex $v_j$ in $V_2$ do
 4:     if $label = false$ or label$(v_i)$ = label$(v_j)$then
 5:       $sim_{ij} \leftarrow 1$
 6:     else
 7:       $sim_{ij} \leftarrow 0.5$
 8: $(tmp_{ij}) = (sim_{ij})$
 9: while $\forall$ indices $i, j : |sim_{ij} - tmp_{ij}| < \epsilon$ do
10:   $(tmp_{ij}) = (sim_{ij})$
11:   for vertex $v_i$ in $V_1$ do
12:     for vertex $v_j$ in $V_2$ do
    //compute parent neighborhood similarity
13:       matrix $(in_{lk}) \in \mathbb{R}^{|p_{G_1}(v_i)| \times |p_{G_2}(v_j)|}$
14:       for row index $0 \leq l < |p_{G_1}(v_i)|$ do
15:         for column index $0 \leq k < |p_{G_2}(v_j)|$ do
16:           row index $l' \leftarrow l^{th}$ index in $p_{G_1}(v_i)$
17:           col. index $k' \leftarrow k^{th}$ index in $p_{G_2}(v_j)$
18:           $in_{lk} \leftarrow (1 - tmp_{l'k'})/\epsilon$
    //search optimal matching
19:       vertex matching $mp \leftarrow$ hungarian$((in_{lk}))$
20:       $sim_{in} \leftarrow \frac{\sum_{l \leftarrow 0}^{\min(|p_{G_1}(v_i)|, |p_{G_2}(v_j)|)-1} in_{lmp_l}}{max(|p_{G_1}(v_i)|, |p_{G_2}(v_j)|)}$
    //compute child neighborhood similarity
21:       matrix $(out_{lk}) \in \mathbb{R}^{|c_{G_1}(v_i)| \times |c_{G_2}(v_j)|}$
22:       for row index $0 \leq l < |c_{G_1}(v_i)|$ do
23:         for column index $0 \leq k < |c_{G_2}(v_j)|$ do
24:           row index $l' \leftarrow l^{th}$ index in $c_{G_1}(v_i)$
25:           col. index $k' \leftarrow k^{th}$ index in $c_{G_2}(v_j)$
26:           $out_{lk} \leftarrow (1 - tmp_{l'k'})/\epsilon$
    //search optimal matching
27:       vertex matching $mc \leftarrow$ hungarian$((in_{lk}))$
28:       $sim_{out} \leftarrow \frac{\sum_{l \leftarrow 0}^{\min(|c_{G_1}(v_i)|, |c_{G_2}(v_j)|)-1} out_{lmc_l}}{max(|c_{G_1}(v_i)|, |c_{G_2}(v_j)|)}$
29:       if $label = false$ then
30:         $sim_{ij} \leftarrow \frac{sim_{in} + sim_{out}}{2}$
31:       else
32:         $sim_{ij} \leftarrow \sqrt{tmp_{ij} \cdot \frac{sim_{in} + sim_{out}}{2}}$
    //search optimal vertex matching
33: vertex matching $m \leftarrow$ hungarian$((sim_{ij}))$
    //similarity computation
34: if $sg = false$then
35:   return $s \leftarrow \frac{\sum_{i \leftarrow 0}^{\min(|V_1|, |V_2|)-1} sim_{im_i}}{\max(|V_1|, |V_2|)}$
36: else
37:   return $s \leftarrow \frac{\sum_{i \leftarrow 0}^{\min(|V_1|, |V_2|)-1} sim_{im_i}}{|V_1|}$
```

Note that a computational complexity of the neighbour matching algorithm is non-trivial as the similarity value depends on the convergence for a given precision which depends on the graph itself, and thus is out of the scope of this work.

In our implementation, we also return the vertex matching m ㉝ to analyze similar vertices in both graphs. To detect multiple matchings, we again execute the algorithm but we initialize the similarity of already matched vertices to 0. Furthermore, we implemented all *for* loops in a parallel fashion to speed up execution.

3.6 Multiresolutional Spectral Analysis

Next, we present our novel graph similarity heuristic based on spectral analysis of adjacency matrices. Our key idea is that eigenvalues of adjacency matrices exhibit two important properties from spectral graph theory [19]:

1) If eigenvalues of two adjacency matrices are different, the graphs are different. This observation does not imply that different graphs have different eigenvalues of their adjacency matrices, so we expect that this only happens with small probability for typical graphs of interest.
2) Eigenvalues are invariant under cyclic permutation with respect to vertex labels. This is an important feature, since we only want to compare the graph topology (and obviously not its vertex labels).

In contrast to other related works on similarity measures based on spectral analysis (e.g., Crawford et al. [20]), we are mainly interested in localized (non-global) similarity information to identify small modules (e.g., a hardware Trojan) in a potentially erroneous graph. Therefore, we use a multiresolutional strategy to search at all local positions.

Notation. Let A be the adjacency matrix of a graph $G = (V, E)$. Moreover, let $\lambda_i, \overrightarrow{v}_i, i = 1, \ldots, |V|$ be the eigenvalues arranged in decreasing order which form the spectral decomposition of A, i.e., $A_i \overrightarrow{v}_i = \lambda_i \overrightarrow{v}_i, i = 1, \ldots, |V|$. Once again, G_1 is the small reference subgraph and G_2 the large targeted one.

Algorithm 3. Spectral Analysis

```
Input: Graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, Integer $k$
Output: Spectral distance matrix $(d_i j) \in \mathbb{R}^{k \times |V_2|}$ for $G_1$ and $G_2$
    //generate local $k$-subgraphs
 1: list of subgraphs $\mathcal{S}_1 \leftarrow \emptyset$
 2: for vertex $v \in$ ranked_vertices$(G_1)$ do
 3:   $\mathcal{S}_1$.append(local_subgraph$(G_1, v, k)$)
 4: list of subgraphs $\mathcal{S}_2 \leftarrow \emptyset$
 5: for vertex $v \in \mathcal{V}_2$ do
 6:   $\mathcal{S}_2$.append(local_subgraph$(G_2, v, k)$)
    //compute spectral distance matrix
 7: row index $i \leftarrow 0$
 8: for subgraph $s_1 \in \mathcal{S}_1$ do
 9:   vector $(\lambda_1^{s_1}, \ldots, \lambda_m^{s_1}) \leftarrow$ eigenvalues$(s_1)$
10:   column index $j \leftarrow 0$
11:   for subgraph $s_2 \in \mathcal{S}_2$ do
12:     vector $(\lambda_1^{s_2}, \ldots, \lambda_n^{s_2}) \leftarrow$ eigenvalues$(s_2)$
13:     spectral distance $d_{ij} \leftarrow \sum_{k \leftarrow 1}^{\max(m,n)} \left| \frac{\lambda_k^{s_2}}{\lambda_1^{s_2}} - \frac{\lambda_k^{s_1}}{\lambda_1^{s_1}} \right|$
14:     $j \leftarrow j + 1$
15:   $i \leftarrow i + 1$
16: return $(d_{ij})$
```

Algorithm 3 shows our graph similarity approach based on multiresolutional spectral analysis. First, we generate local k-subgraphs (❶ - ❻) for G_1 and G_2. Since a cross-comparison of all k-subgraphs for both G_1 and G_2 is computationally expensive, we limit the number of k-subgraphs for G_1. To this end, we make the following assumption: if the small subgraph G_1 is present in G_2, this should turn up in a comparison for basically any vertex of G_1. Hence, we select some *representative* vertices in G_1 (ranked_vertices in ❷ (e.g., determined by Google's page rank algorithm [21]). Second, we compute the spectral distance matrix (❼-⓯). In particular, we compute eigenvalue vectors of the subgraphs s_1 and s_2 by means of the function eigenvalue (❾ and ⓬). Note that we assume that eigenvalues are arranged in decreasing order, i.e., λ_1 is the largest eigenvalue. We then compute the spectral distance with normalized eigenvalue sequences ⓭. Finally, matching vertices in G_2 are identified by the smallest spectral distance to any of the vertices in G_1. The computational complexity of our spectral analysis is bound by $\mathcal{O}(\binom{|\mathcal{V}_2|}{k})$ (for a complete graph) since the number of subgraphs in $\mathcal{S}_1$ is bound by the number of ranked vertices, i.e., $k = 3$ in our evaluation. Moreover note that the eigenvalue computation for a $k \times k$ matrix is neglected as a constant factor since k is small.

Note that our spectral analysis returns a spectral distance matrix rather than a rational similarity value when compared to aforementioned algorithms. Since our approach analyzes a spectral distance, distance values of 0 indicate a high similarity.

4 Evaluation

We now provide results of our three case studies, namely gate-level netlist reverse engineering (Section 4.2), Trojans detection (Section 4.3), and obfuscation assessment (Section 4.4). In addition, we provide implementation-specific details for previously mentioned graph similarity algorithms.

4.1 Implementation

Graph Similarity Algorithms. We implemented the graph edit distance approximation (Section 3.4) and neighbor matching (Section 3.5) in C++ 14 using Boost for graph processing, BuDDy for Binary Decision Diagrams (BDDs), munkres-cpp for the *Hungarian* method, and *Nauty* for graph canonicalization. In particular, we used OpenMP for parallelization to significantly accelerate computation, see Section 3 for the steps that can be parallelized for each algorithm. The spectral analysis (Section 3.6) is implemented in R and Python. For our experiments, we utilized several Google Cloud Platform instances with 64 vCPUs which costs around \$3/h per instance.

Gate-Level Netlist Generation. We generated gate-level netlists for numerous designs using the Xilinx Synthesis Technology (XST) suite from Xilinx ISE 14.7, see Table 2. Our evaluation targets 3 Xilinx FPGA families, namely Spartan-6 (xc6slx16), Virtex-6 (xc6vlx75t), and Kintex-7 (xc7k70t). Furthermore, we consider the available XST synthesis optimization goals *speed*, and *area* to measure the robustness among different synthesis options. We want to emphasize that we also evaluated each case study for other families and different FPGA devices within the same families yielding similar results. In addition to the aforementioned preprocessing techniques in Section 3.2, we also removed all (for our case irrelevant) buffers from each analyzed gate-level netlist to reduce the graph size and thus computation time.

TABLE 2
Hardware Design Description and Resource Utilization Synthesized for xc6slx16 With Optimization Goal *Area*

Design	Description	Source	#LUTs	#FFs	Freq. (MHz)
0	Composite-field Sbox	[22]	63	0	-
1	AES (composite-field Sbox)	[22]	2049	587	225
2	AES (table-based Sbox)	[22]	781	584	162
3	AES (table-based Sbox)	[22]	1006	586	177
4	AES (table-based Sbox)	[23]	5101	723	91
5	AES (PPRM-based Sbox)	[22]	2230	587	91
6	AES (ANF-based Sbox)	[22]	6469	587	89
7	AES (table-based Sbox)	[24]	517	692	125
8	I^{2}C Bus	[25]	293	154	214
9	12-bit PIC CPU	[26]	514	248	64
10	16-bit MSP430 CPU	[27]	2235	686	43
11	AES (no Trojan)	[28]	1917	1077	181
12	AES (with Trojan)	[28]	1992	1162	181
13	Trojan (in design 12)	[28]	48	1	-
14	GCD (no obfuscation)	[29]	234	128	152
15	GCD (with obfuscation)	[29]	325	96	171

We selected xc6slx16 FPGA as an representative, since resource utilization only slightly *deviate for other FPGA families.*

We want to remark that we additionally evaluated reliability against potential errors in the netlist graphs (occurring due to imperfect image processing in chip-level reverse engineering). To this end, we randomly changed or deleted around 5 percent of all edges, yielding similar results. Moreover, we investigated whether grouping of nets (bundled wires connecting one or more gates) had an influence on accuracy, however, we observed that the accuracy is only marginally effected decimal places of the similarity value. Thus, we deliberately omitted this in both algorithm's description and practical evaluation.

4.2 Case Study I: Netlist Reverse Engineering

In our first case study, we evaluated the use of graph similarity algorithms for gate-level netlist reverse engineering with a particular focus on security-critical circuits. To this end, we examined to what extent graph similarity algorithms can reliably identify specific parts of a cryptographic primitive, i.e., identification an Sbox implementation based on a composite-field optimization as part of the widely-used cipher Advanced Encryption Standard (AES). We want to emphasize that this case study represents a typical instance of gate-level netlist reverse engineering, since we focus on identification of submodules in a flattened netlist which subsequently enables to retrieve hierarchy information and parts of the original high-level design implementation goals. In addition, knowledge about the internal architecture of a cryptographic design provides valuable information for other scenarios, for example, to enable injection of cryptographic Trojans through Sbox tampering or to improve assessment of physical attacks such as fault injection or side-channel analysis [12].

4.2.1 Hardware Designs

We obtained numerous publicly available third-party AES implementations and other hardware designs such as CPUs (e.g., from OpenCores). Note that our considered AES

TABLE 3
Gate-Level Netlist Reverse Engineering Case Study Results (Phase 1)
Comparison Between Design 0 (Composite-Field AES Sbox) and Designs 1 - 10

Device	Synthesis Option	Algorithm	Hardware Design									
			1	2	3	4	5	6	7	8	9	10
XC6SLX16	*speed*	GED	0.921	0.822	0.779	0.896	0.945	0.908	0.705	0.768	0.826	0.952
XC6SLX16	*area*	GED	0.915	0.763	0.768	0.898	0.942	0.903	0.704	0.719	0.791	0.942
	Computation Time		3.22s	1.81s	2.66s	10.8s	3.44s	19.0s	0.87s	0.30s	2.31s	25.1s
XC7K70T	*speed*	GED	0.944	0.766	0.758	0.900	0.945	0.912	0.695	0.707	0.799	0.951
XC7K70T	*area*	GED	0.950	0.762	0.768	0.896	0.953	0.915	0.695	0.719	0.791	0.942
	Computation Time		3.51s	1.74s	3.32s	10.7s	3.77s	20.1s	1.00s	0.29s	2.60s	29.7s
XC6VLX75T	*speed*	GED	0.910	0.763	0.758	0.897	0.942	0.920	0.695	0.748	0.799	0.950
XC6VLX75T	*area*	GED	0.922	0.762	0.768	0.898	0.941	0.916	0.695	0.719	0.791	0.943
	Computation Time		3.61s	1.81s	3.37s	9.15s	4.26s	19.4s	0.79s	0.26s	2.27s	24.7s
XC6SLX16	*speed*	NM	0.770	0.610	0.571	0.699	0.779	0.783	0.526	0.615	0.641	0.773
XC6SLX16	*area*	NM	0.763	0.567	0.573	0.665	0.752	0.757	0.541	0.575	0.622	0.745
	Computation Time		54.7s	26.9s	36.9s	4.01m	59.3s	10.7m	14.2s	6.96s	36.6s	4.05m
XC7K70T	*speed*	NM	0.805	0.578	0.564	0.710	0.783	0.774	0.425	0.583	0.629	0.782
XC7K70T	*area*	NM	0.809	0.567	0.573	0.665	0.757	0.765	0.420	0.576	0.622	0.754
	Computation Time		49.6s	11.8s	28.6s	3.75m	58.5s	9.18m	7.84s	5.33s	30.7s	4.16m
XC6VLX75T	*speed*	NM	0.755	0.579	0.564	0.708	0.786	0.778	0.425	0.605	0.629	0.780
XC6VLX75T	*area*	NM	0.773	0.567	0.573	0.665	0.758	0.768	0.420	0.576	0.622	0.749
	Computation Time		53.1s	16.1s	30.5s	3.98m	7.84s	8.22m	14.1s	7.12s	36.5s	3.83s

GED - *Graph edit distance approximation*
NM - *Neighbour matching using* $\epsilon = 0.0001$
AES Sbox is synthesized for XC6SLX16 with optimization goal area. *Parameter* subgraph *and* label *are* true *for all experiments, and only the combinational logic subgraph preprocessing technique is used.*

designs (1 - 7) utilize different Sbox implementation strategies (e.g., precomputed lookup tables or composite-fields) [30]. To demonstrate the reliability of graph similarity algorithms, we provide results for other non-cryptographic general-purpose designs, i.e., design 8 (I^2C), design 9 (12-bit PIC CPU), and design 10 (MSP430 CPU). Table 2 provides further details for each hardware design such as resource consumptions and origins of each design.

We want to emphasize that design 1 (composite-field Sbox implementation) should exhibit highest similarity for all designs in this case study, since the composite-field Sbox of this design is our small reference subgraph G_1. Also, we expect design 5 (PPRM-based Sbox implementation) and design 6 (ANF-based Sbox implementation) to cause *high* similarity values, since both Sbox implementation strategies are mainly based on AND-XOR gates. In addition, note that both phase 1 and phase 2 are used in this case study.

4.2.2 Results (Phase 1)

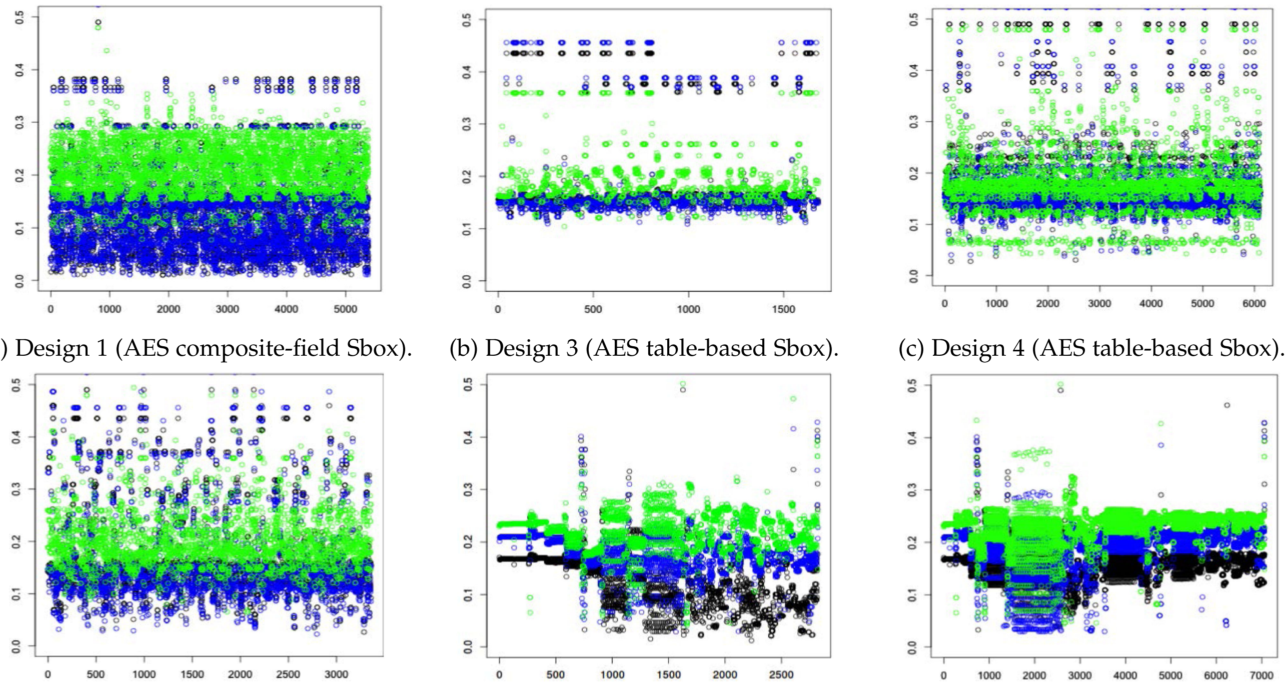
Our evaluation results for phase 1 analysis (combinational logic subgraph) for the similarity comparison of the composite-field Sbox to other designs are summarized in Table 3 and Fig. 2.

Table 3 shows results for graph edit distance approximation and neighbour matching for designs 1 - 10 for three Xilinx FPGA families and both synthesis optimization goals *speed* and *area*. The graph edit distance approximation indicates a high similarity ≥ 0.9 to the composite-field Sbox for designs 1, 4, 5, 6, and 10 independent of FPGA family or optimization goal. The neighbour matching indicates a high similarity ≥ 0.77 for designs 1, 5, 6, and 10 independent of FPGA family or optimization goal.

Fig. 2 shows results for several representative designs for our multiresolutional spectral analysis. Similar to graph edit distance and neighbour matching, we see that distance matrices indicate a high similarity for design 1 since it has sufficiently small spectral distances (< 0.05). Similar to graph edit distance and neighbor matching, we see that designs 5, 6, and 10 exhibit some similarity to the composite-field Sbox. Note that we selected three vertices after ranking: (1) top-ranked vertex (marked in black), (2) 75 percent-quantile vertex (marked in blue), and (3) 50 percent-quantile vertex (marked in green). Moreover, the top-10 candidate vertices with smallest spectral distance to the three chosen vertices are actual S-Box vertices with accuracy of 100 percent (top-ranked), 100 percent (for 75 percent-quantile), and 70 percent (for 50 percent-quantile). Thus, a statistical test shows that 27 out of 30 true-positive vertices rejects a null hypothesis with high significance (binomial test, p-value $< 10^{-7}$). Note that the results for different synthesis options and FPGA families are similar to the one in Fig. 2, hence we deliberately deliberately did not provide evaluation figures.

Hence, graph edit distance, neighbor matching, and spectral analysis yield high similarities to designs 1, 5, 6, and 10, since these three similarity algorithms provide an accurate and reliable measure. Note that similarity scores are determined within seconds to minutes for all algorithms.

We want to emphasize that the combinational logic subgraph which exhibits highest similarity for design 10 is a register using 81 FFs handling memory access. Obviously, such a register size is implausible for an AES implementation, hence a reverse engineer would put this result aside in practice and just focus on designs 1, 5, and 6 since these Sbox candidate gates are identified for a subgraph with



(a) Design 1 (AES composite-field Sbox). (b) Design 3 (AES table-based Sbox). (c) Design 4 (AES table-based Sbox).

(d) Design 10 (MSP430 processor). (e) Design 5 (AES PPRM-based Sbox). (f) Design 6 (AES ANF-based Sbox).

Fig. 2. Gate-level netlist reverse engineering case study results for our multiresolutional spectral analysis without any preprocessing techniques, comparison between design 0 (composite-field AES Sbox) and designs 1 - 10. AES Sbox (composite field) is synthesized for xc6slx16 with optimization goal *area*, the representative designs in a) - f) are also synthesized for xc6slx16 with optimization goal *area*. The top-ranked vertex (marked in black), 75 percent-quantile vertex (marked in blue), and 50 percent-quantile vertex (marked in green). The y-axis shows the spectral distance, and the x-axis shows the vertex labels.

128-bit, i.e., the actual AES datapath registers. Note that our graph similarity analysis yields 63 gates that have to be further analyzed (e.g., with Boolean function analysis or manually).

Sensitivity of Parameter Choice. In addition to aforementioned results, we present several counterexamples which demonstrate why our selected parameters and preprocessing techniques perform best.

If we compute similarity of the composite-field Sbox to design 1 using GED without any preprocessing technique and parameters *subgraph* = *label* = *false*, we obtain a similarity value of 0.0035 in $5m$ (*baseline*). Enabling both parameters *subgraph* = *label* = *true* yields a high similarity value of 0.943 in $2s$, however, the matched gates in design 1 contain around 10 percent registers which are erroneously matched to combinational logic gates of the Sbox circuit. To this end, our combinational logic subgraph preprocessing generally prevents this mismatch between combinational and synchronous gates and thus increase accuracy.

If we compute similarity of the composite-field Sbox to design 1 (AES using the composite-field Sbox) using GED algorithm in phase 1, *subgraph* = *false*, and *label* = *true*, we obtain a *low* similarity value of 0.239 in $15s$ (compared to the *high* similarity value of 0.915 for *subgraph* = *true*). Even though GED determines correct Sbox gates in design 1, a *low* similarity value occurs due to the original similarity computation equation, see Section 3.4.

If we compute similarity of the composite-field Sbox to design 1 with all other AES designs (design 2 - 7) using GED algorithm in phase 1, *subgraph* = *true*, and *label* = *false*, we obtain *high* similarity values ranging between 0.963 and 1.000. Thus without labeled vertices check, similarity values cannot be used for effective distinction.

4.2.3 Results (Phase 2)

Since phase 1 analysis indicates a high (false-positive) similarity score design 10, we now provide results of our more robust phase 2 (LUT decomposition and combinational logic bitslice) similarity analysis. Note that our LUT decomposition unifies different FPGA families, hence we deliberately selected Spartan-6 as a representative as other families yield similar results. Also, we selected graph edit distance approximation since its requires the least computation time in phase 1.

Our evaluation results are summarized in Table 4. We see that graph edit distance approximation indicates a high similarity ~ 0.9 to bitslices of the composite-field Sbox while design 10 exhibits a similarity of ~ 0.8, thus we have

TABLE 4
Gate-Level Netlist Reverse Engineering Case Study Results (Phase 2), Comparison Between Design 0 and 1, 5, 6, and 10

Device	Synthesis Option	Algorithm	Hardware Design			
			1	5	6	10
xc6slx16	*speed*	GED	0.899	0.914	0.885	0.818
xc6slx16	*area*	GED	0.899	0.899	0.887	0.818
	Computation Time		10.3h	11.4h	11.5h	68h

AES Sbox (composite field) is synthesized for xc6slx16 *with optimization goal* area. *Parameter* subgraph *and* label *are* true *for all experiments, and the combinational logic bitslice and LUT decomposition preprocessing techniques are used.*

TABLE 5
Trojan Detection Case Study Results (Phase 1) Comparison Between Design 13 and Designs 1, 5 - 12

Device	Synthesis Option	Algorithm	Hardware Design								
			11	12	1	5	6	7	8	9	10
xc6slx16	*speed*	GED	0.779	1.000	0.699	0.696	0.753	0.702	0.867	0.865	0.958
xc6slx16	*area*	GED	0.722	1.000	0.699	0.691	0.699	0.699	0.870	0.865	0.945
	Computation Time		28.0s	29.8s	2.53s	2.51s	13.0s	0.74s	0.19s	1.87s	24.7s
xc7k70t	*speed*	GED	0.779	1.000	0.696	0.691	0.751	0.702	0.808	0.865	0.958
xc7k70t	*area*	GED	0.715	1.000	0.699	0.699	0.702	0.699	0.870	0.865	0.943
	Computation Time		8.02s	9.65s	2.70s	2.58s	14.9s	0.69s	0.19s	2.04s	25.0s
xc6vlx75t	*speed*	GED	0.772	1.000	0.696	0.694	0.756	0.702	0.865	0.865	0.958
xc6vlx75t	*area*	GED	0.753	1.000	0.699	0.699	0.699	0.699	0.870	0.865	0.945
	Computation Time		6.72s	9.73s	2.67s	2.42s	12.7s	0.71s	0.21s	1.94s	26.2s
xc6slx16	*speed*	NM	0.692	0.825	0.630	0.615	0.670	0.474	0.691	0.614	0.767
xc6slx16	*area*	NM	0.639	0.825	0.619	0.592	0.647	0.475	0.681	0.609	0.811
	Computation Time		2.08m	3.05m	25.6s	31.8s	10.4m	7.42s	3.84s	19.5s	2.85m
xc7k70t	*speed*	NM	0.595	0.825	0.631	0.615	0.669	0.468	0.642	0.610	0.766
xc7k70t	*area*	NM	0.541	0.825	0.627	0.611	0.645	0.471	0.681	0.609	0.811
	Computation Time		45.5s	1.06m	30.3s	34.8s	8.75m	7.32s	3.12s	19.8s	2.67m
xc6vlx75t	*speed*	NM	0.580	0.825	0.629	0.618	0.669	0.468	0.689	0.610	0.762
xc6vlx75t	*area*	NM	0.541	0.825	0.618	0.615	0.645	0.471	0.681	0.609	0.811
	Computation Time		43.6s	1.02m	27.9s	36.3s	6.53m	8.09s	2.56s	19.2s	2.58m

GED - *Graph edit distance approxmiation*
NM - *Neighbour matching using* $\epsilon = 0.0001$
Trojan (design 13) is synthesized for xc6slx16 *with optimization goal* area. *Parameter* subgraph *and* label *are* true *in all experiments, and only the combinational logic subgraph preprocessing technique is used.*

evidence that the composite-field Sbox is unlikely present in design 10. Hence, we have a certain degree of confidence that a composite-field Sbox gate structure is within designs 1, 5, and 6, but not in design 10.

In summary, we demonstrated that graph similarity algorithms can indeed be utilized for automated and reliable gate-level netlist reverse engineering. To this end, graph edit distance approximation, neighbor matching, and spectral analysis should be used in concert to report reliable and accurate similarity. In case phase 1 analysis yields high similarities for more than one design, phase 2 analysis should be used to obtain more reliable and accurate similarity results. We want to emphasize that we are the first to demonstrate automatic reverse engineering of composite-field-based Sboxes to the best of our knowledge. So far it was only demonstrated that precomputed LUT Sboxes can be automatically identified in third-party IP cores [31].

4.3 Case Study II: Trojan Detection

Over the past decade, numerous works have addressed th emerging threat of hardware Trojans since current IC design and fabrication practices rely on untrusted entities (e.g., untrusted third-party IP cores or untrusted offshore fab). To counteract this threat and inspired by malicious software detection approaches [9], we evaluated whether graph-similarity algorithms can be leveraged to reliably detect hardware Trojans in gate-level netlists of potentially untrusted third-party IP cores.

4.3.1 Hardware Designs

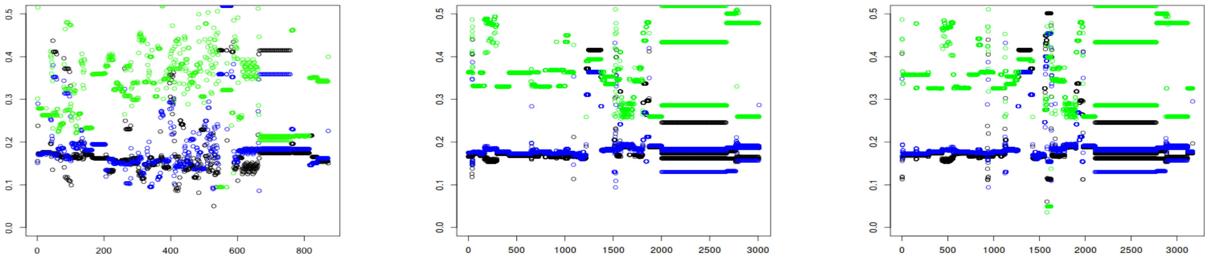
We obtained a publicly available hardware Trojan benchmark AES-T1000 from the trusthub benchmark suite [32]. Design 11 refers to the AES-T1000 without the Trojan, design 12 refers to the AES-T1000 including the Trojan, and design 13 is the Trojan itself. The Trojan leaks the AES key for a predefined input plaintext through a covert power side-channel using a code-division multiple access sequence. More specifically, an Linear Feedback Shift Register (LFSR)-based Pseudo Random Number Generator (PRNG) (initialized with the input plaintext) is used to XOR modulate the secret key and finally the output of the XOR gate is connected to 8 identical FF gates to mimic a large capacitance.

To demonstrate the reliability of graph similarity algorithms, we also provide results whether the Trojan designs exhibits *high* similarities to other benign designs, i.e., design 1, 5, 6, and 7 (AES circuits using different Sbox implementation strategies), design 8 (I^2C), design 9 (12-bit PIC), and design 10 (MSP430). Table 2 provides resource consumptions for each hardware design.

We want to emphasize that design 12 (AES-T1000 including the Trojan) should exhibit highest similarity to the Trojan for all designs in this case study.

4.3.2 Results (Phase 1)

Our evaluation results are summarized in Table 5 and Fig. 3. Table 5 shows results for graph edit distance approximation and neighbour matching, for designs 1, 5, 6, 7, 8, 9, 10, 11, and 12 for three Xilinx FPGA families and both synthesis optimization goals *speed* and *area*. The graph edit distance approximation indicates highest similarity 1.0 for design 12 independent of FPGA family or optimization goal, followed by the MSP430 processor with ~ 0.95. The neighbour matching also indicate a high similarity ≥ 0.852 for design 12 independent of FPGA family or optimization goal. Fig. 2 shows results for several representative designs



(a) Design 9 (12-bit PIC). (b) Design 11 (AES without Trojan). (c) Design 12 (AES with Trojan).

Fig. 3. Trojan detection case study results for our multiresolutional spectral analysis without any preprocessing techniques, comparison between design 13 and designs 9, 11, and 12. Trojan (design 13) is synthesized for xc6slx16 with optimization goal *speed*, the representatative designs 9, 11, and 12 are also synthesized for xc6slx16 with optimization goal *area*. The top-ranked vertex (marked in black), 75 percent-quantile vertex (marked in blue), and 50 percent-quantile vertex (marked in green). The y-axis shows the spectral distance, and the x-axis shows the vertex labels.

for our multiresolutional spectral analysis. We selected three vertices after ranking: (1) top-ranked vertex (marked in black), (2) 75 percent-quantile vertex (marked in blue), and (3) 50 percent-quantile vertex (marked in green). Based on results in Table 6, we selected designs 9 - 12, and similar to graph edit distance approximation and neighbour matching, spectral analysis does identify a higher similarity to the hardware Trojan for design 12 and less similarity to design 9 and 11. Note that vertices with the smallest distance to the 50 percent-quantile vertex (marked in green) in design 12 belong to the actual Trojan. Since we have underpinned robustness of our spectral analysis regarding different FPGA families and optimization goals, see Section 4.2, we deliberately omit these results here.

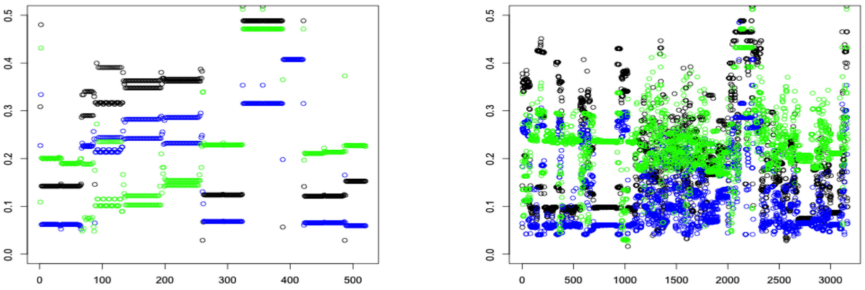
Note that similarity scores are determined within seconds to minutes for all algorithms. Furthermore, other parameter configurations for graph similarity algorithms and preprocessing techniques yield similar negative results, see Section 4.2.2. Since our evaluation results indicate that design 12 is most similar design for phase 1 analysis (and we actually identify the hardware Trojan), we deliberately not evaluate phase 2 results. We want to note that other parameter configurations for the graph similarity algorithms and preprocessing techniques yield similar negative results, see Section 4.2.2.

In summary, we demonstrated that graph similarity can indeed be utilized for automated and reliable hardware Trojan detection in untrusted third-party IP cores. To this end, graph edit distance approximation, neighbor matching, and our spectral analysis should be used in concert to report accurate and reliable similarity values for hardware Trojan detection. Additionally we want to note that other static hardware Trojan detection schemes (such as FANCI or COTD [33]) are able to detect the selected Trojan, however, our goal was to demonstrate that known Trojan can also be identified with graph similarity based approaches.

TABLE 6
Hardware Obfuscation Assessment Case Study Results (Phase 1) Comparison Between Design 15 and 1, 5 - 10, 14

Device	Synthesis Option	Algorithm	Hardware Design							
			14	1	5	6	7	8	9	10
xc6slx16	*speed*	GED	0.975	0.741	0.754	0.759	0.643	0.816	0.841	0.926
xc6slx16	*area*	GED	0.979	0.738	0.749	0.750	0.000	0.808	0.831	0.937
		Computation Time	2.52s	11.8s	14.3s	66.1s	1.93s	1.91s	16.6s	2.51m
xc7k70t	*speed*	GED	0.975	0.755	0.756	0.764	0.000	0.000	0.840	0.925
xc7k70t	*area*	GED	0.979	0.750	0.750	0.751	0.000	0.808	0.831	0.942
		Computation Time	3.46s	13.1s	14.2s	72.7s	1.67s	0.44s	15.5s	2.56m
xc6vlx75t	*speed*	GED	0.975	0.734	0.754	0.762	0.000	0.801	0.839	0.926
xc6vlx75t	*area*	GED	0.979	0.728	0.750	0.750	0.000	0.808	0.831	0.942
		Computation Time	3.45s	13.3s	13.3s	63.8s	1.61s	1.87s	16.3s	2.56m
xc6slx16	*speed*	NM	0.973	0.624	0.627	0.687	0.000	0.584	0.573	0.717
xc6slx16	*area*	NM	0.970	0.603	0.611	0.680	0.000	0.532	0.566	0.692
		Computation Time	22.2s	6.11m	6.58m	44.3s	1.21s	23.2s	2.28m	24.9m
xc7k70t	*speed*	NM	0.973	0.623	0.627	0.687	0.000	0.000	0.570	0.712
xc7k70t	*area*	NM	0.970	0.613	0.613	0.682	0.000	0.531	0.566	0.686
		Computation Time	10.8s	4.96m	5.08m	37.1m	0.89s	19.4s	2.21m	23.5m
xc6vlx75t	*speed*	NM	0.973	0.607	0.649	0.688	0.000	0.522	0.570	0.710
xc6vlx75t	*area*	NM	0.970	0.610	0.627	0.683	0.000	0.531	0.566	0.688
		Computation Time	9.58s	5.90m	6.08m	36.3m	1.32s	18.8s	2.25m	23.2m

GED - *Graph edit distance approxmiation*
NM - *Neighbour matching using* $\epsilon = 0.0001$
GCD (design 15) is synthesized for xc6slx16 with optimization goal area. *Parameter* subgraph *and* label *are* true *and only the combinational logic subgraph preprocessing technique is used.*



(a) Design 14 (GCD with obfuscation). (b) Design 10 (MSP430 processor).

Fig. 4. Hardware obfuscation assessment case study results for our multiresolutional spectral analysis without any preprocessing techniques, comparison between design 15 and designs 10 and 14. GCD (design 15) is synthesized for xc6slx16 with optimization goal *area*, the designs 14 and 10 are also synthesized for xc6slx16 with optimization goal *area*. The top-ranked vertex (marked in black), 75 percent-quantile vertex (marked in blue), and 50 percent-quantile vertex (marked in green). The y-axis shows the spectral distance, and the x-axis shows the vertex labels.

4.4 Case Study III: Obfuscation Assessment

Over the past decades numerous hardware obfuscation transformations have been developed to protect valuable IP from reverse engineering and other threats, see Shakaya et al. [5] for a comprehensive overview. However, the development of practical and sound obfuscation schemes is challenging since metrics for obfuscation are hard to quantify (e.g., how much effort has to be invested to break obfuscation) since it requires to quantify a human reverse engineer which is challenging and still unsolved so far [34].

In our third case study, we evaluate the application of graph similarity analysis for assessment of obfuscation transformations. We want to emphasize that the ideal goal of an obfuscation transformation is to destroy any relation between an unobfuscated circuit C_1 and its obfuscated version $\mathcal{O}(C_1)$, so a graph similarity analysis of C_1 and $\mathcal{O}(C_1)$ should not yield a higher similarity than for other circuits $C_2, \ldots, C_n$ implementing a different functionality. Otherwise (if there is a significant similarity of C_1 to $\mathcal{O}(C_1)$), we can derive critical information from an obfuscated circuit and thus circumvent the obfuscation. We acknowledge that graph similarity analysis is obviously not sufficient to entirely quantify a degree of obfuscation, however, it supports obfuscation designers with a valuable metric indicating topological difference induced by a transformation.

Hardware Obfuscation Transformations. In order to demonstrate a typical obfuscation assessment, we selected the obfuscation transformation proposed by Li et al. [29] targeting obfuscation of sequential circuits. In particular, we reimplemented the *conditional stuttering* and *sweep* transformations for the 32-bit Greatest Common Divisor (GCD) circuits as proposed.

Note that we deliberately did not choose Finite State Machine (FSM)-based obfuscation transformations (e.g., [35]), since they do not or marginally alter a circuit's datapath and thus do not induce large topological differences. Hence, it is obvious that our approach works and detects datapath circuits such as an Sbox, see Section 4.2.

4.4.1 Hardware Design

To assess the obfuscation scheme, we evaluate the two GCD circuits (designs 14 - unobfuscated and 15 - obfuscated). Moreover, we selected several cryptographic designs (design 1, 5, 6, 7) and general-purpose hardware design: design 8 (I^2C), design 9 (12-bit PIC), and design 10 (MSP430). In Table 2 we provide resource consumptions for each hardware design.

We want to emphasize that design 14 (unobfuscated) should exhibit highest similarity for all designs in this case study, since we compare all designs with the obfuscated GCD circuit (design 15).

4.4.2 Results (Phase 1)

Our evaluation results are summarized in Table 6 and Fig. 4. Table 6 shows results for graph edit distance approximation and neighbour matching. Overall, we see that graph edit distance approximation and neighbour matching indicate design 14 as *highly* similar to the obfuscated GCD (similarity values $[0.97, 1.0]$), independent of FPGA family or synthesis optimization goal. Note that the reason for this unreliability is similar to the one described in case study 1. Moreover note that all combinational subgraphs for 7 (tiny AES implementation) and 8 (I^2C bus) (for the 0.000 results) are smaller than the smallest one for design 15, and hence design 15 cannot be part of designs 7 or 8 in these cases.

Fig. 4 shows results for our spectral analysis. Based on results in Table 6, we selected design 14 and 10 as representatives since design 10 exhibits also high similarities for graph edit distance approximation. We selected three vertices after ranking: (1) top-ranked vertex (marked in black), (2) 75 percent-quantile vertex (marked in blue), and (3) 50 percent-quantile vertex (marked in green). Similar to graph edit distance approximation, both design exhibit *high* similarities since both spectral distance matrices possess vertices with distance $[0.5, 0.7]$. Since we have underpinned the robustness of our spectral analysis regarding different FPGA families and optimization goals, see Section 4.2, we deliberately omit these results here.

Similarity scores are determined within seconds to minutes for all algorithms. Furthermore, other parameter configurations for graph similarity algorithms and preprocessing techniques yield similar negative results, see Section 4.2.2. Since our evaluation results indicate design 14 as most similar design for phase 1 analysis, we deliberately do not further investigate phase 2. We want to note that other parameter configurations for graph similarity algorithms and preprocessing techniques yield similar negative results, see Section 4.2.2.

In summary, we demonstrated graph similarity algorithms provide a valuable metric to indicate an obfuscation degree to support both designers of obfuscation transformations and engineers instantiating them in their designs. For the selected GCD circuit, we see that the topological difference induced by the obfuscation transformation may not be sufficient to hamper reverse engineering. Furthermore, we want to emphasize that this assessment approach scales to larger designs including multiple IP cores since we analyze register groups with our combinational logic subgraph preprocessing. To report a reliable and accurate metric, graph edit distance approximation, neighbor matching and spectral analysis should be used in concert.

5 Discussion

Implications. In previous case studies, we have demonstrated that graph similarity has a variety of applications to hardware security. We have shown that graph similarity heuristics *indeed* provide accurate and reliable *heat spots* while keeping analysis time practical. Our case studies demonstrated that in general graph edit distance approximation, neighbor matching, and spectral analysis should be used in concert (with a majority vote) to report reliable and accurate similarity values, using labeled vertices, subgraph adjustments, and our two-phased analysis. As noted in Section 4.1, we also evaluated other similarity heuristics and subgraph isomorphism algorithms, however, their computation time or accuracy turned out to be impractical for larger graphs.

We want to emphasize that we deliberately did not perform a pair-wise comparison of large designs, since our main focus is to find small modules (e.g., hardware Trojans or datapath circuits) rather than comparing similarity of two large designs.

Algorithm Selection. As noted before graph similarity approaches are used in a wide strand of research fields from bioinformatics to software security research. Since each concrete similarity algorithm exploits graph characteristics for a field-specific problem, we selected algorithms which have already shown to be effective for software security research. Algorithms used for graph similarity problems in the software context are close to our hardware applications (e.g., Trojan detection and reverse engineering), see Chan et al. [18] or Hu et al. [9].

Analysis and evaluation of graph similarity algorithms from other fields (e.g., data mining Li et al. [36], Zhang et al. [37]) based on graph kernels or graphlet comparisons are out of the scope of this work and may be investigated (and potentially adapted to the hardware context) by future research.

Generality. Our approach scales even to larger designs including numerous IP cores, because only the number of combinational logic subgraphs will increase but not their size. Moreover, subgraphs can be processed in parallel. In addition, we want to emphasize that our graph similarity heuristics are not specific to FPGAs and can be applied to Application Specific Integrated Circuit (ASIC) gate-level netlists as well.

Theoretical Limitations. We acknowledge that our work has limitations with respect to statements regarding theoretical bounds or proofs of convergence. Proofs or statement of soundness for presented graph similarity heuristics are highly desirable from a theoretical point of view, however, they are an open challenge and out of scope of our work. For example, this has to consider the distribution of test statistics "*under the alternative*" (non-zero difference between the graphs locally) for our multiresolutional spectral analysis. Moreover, we want to emphasize that in practice, a reverse engineer is interested in accurate and reliable *heat spots* which are determined in practical computation times so that he can investigate the identified subcircuit in more detail.

Future Work. Research on graph similarity can be investigated in several directions. For example similarity analysis might be used to analyze security of split-manufacturing schemes, since isomorphism property might be to rigid for a security criterion as explained before. For further investigations, our evaluation could be extended to a variety of synthesizers to examine reliability among different synthesizers. As noted before, future work may also explore theoretical bounds or proofs of convergence to support statements of similarity algorithms.

6 Related Work

Gate-Level Netlist Reverse Engineering. Modern digital circuit design is typically realized at RTL which models signal flow among registers. Logic synthesis tools convert RTL descriptions to gate-level netlists, i.e., a list of gates and their interconnections. From a reverse engineering perspective, valuable information is lost during this translation: module boundary information as well as hierarchy information [7]. In addition, diverse optimizations are performed to achieve a predefined optimization goal.

Several works targeted automatic functional module extraction (e.g., [7]) such as FSMs or datapath circuits such as adders. Meade et al. [38] performed another notable work using a similarity-based approach, since they examined similarity of a netlist's graph topology to identify control registers of state machines. Note that their technique is based on similarity analysis of one netlist, rather than our technique which compares similarity between two netlists. While previous approaches such as Boolean function analysis or (sub)graph isomorphism require strict information, i.e., an error-free netlist, we investigated graph similarity algorithms which are reliable even in the presence of errors and obfuscation, see Section 1. Note that in case any error (e.g., a misidentified gate type or a missing signal) is present, approaches based on strict Boolean measures such as subgraph isomorphism or Boolean function analyses cannot be applied to yield desired results which emphasizes the general use of similarity approaches. In particular, errors are particularly worrisome in case a malicious Trojan was equipped with physical design obfuscation [4] to evade Boolean function analyses by design.

In addition, similarity analyses can be leveraged during IC design, simulation, verification and testing phases to improve designer's productivity for IP reuse [39]. We want to emphasize that such approaches are orthogonal to reverse engineering since high-level information is already available.

Hardware Trojans. Since an initial report by the US DoD in 2005, the scientific community extensively researched offensive and defensive aspects of hardware Trojans, see Bhunia et al. [3] for a comprehensive overview. In general, a hardware Trojan consists of a *payload* circuit delivering the malicious functionality (e.g., leakage of cryptographic keys or denial of service) and an optional payload activating *trigger* circuit (e.g., a counter or sensor). Defensive research focuses on detection of hardware Trojans based on diverse characteristics such as physical attributes, trigger features, and payload features. In order to detect Trojan's characteristics, various approaches based on side-channel analysis, and design analysis have been proposed. Our work is comparable to static analysis approaches. Hasagawa et al. [40] proposed a hardware Trojan classification based on machine learning using support vector machines. Note that support vector machines and graph similarity are fundamentally different approaches: the first being a technique from supervised learning whereas the second is a descriptive method which yields data on similarity properties and can be used for both supervised or unsupervised methods.

7 Conclusion

Hardware reverse engineering is a general tool for a variety of purposes. On one hand it enables detection of malicious circuitry or find evidence for IP infringement, on the other hand it also reveals necessary high-level information to facilitate malicious circuitry injection. Numerous works addressed this arms race between reverse engineering techniques and obfuscation transformations.

In this paper, we presented the graph similarity problem for the first time in the domain of hardware security, particularly for reverse engineering, Trojan detection, and assessment of obfuscation. To this end, we significantly improved graph similarity heuristics with optimizations tailored to hardware designs. Furthermore, we introduced a new technique based on a multiresolutional spectral graph analysis. In our three case studies, we demonstrated the practical feasibility of graph similarity for different FPGA families and several design optimization goals. Particularly, our results showed that graph edit distance approximation, neighbor matching, and our spectral analysis should be used in concert to report accurate and reliable similarity scores.

Since we believe that our work represents a fundamental building block for future research and applications in industry, we plan to publicly release our implementation.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable feedback and comments. Part of this work was supported by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (ERC Advanced Grant No. 695022 (EPoCH)), by the US National Science Foundation (NSF) award CNS-1563829, by the German Federal Ministry of Education and Research (BMBF Grant 16KIS0820 emproof and MED4D, part project 05M16PCA).

References

[1] M. Rostami, F. Koushanfar, and R. Karri, "A primer on hardware security: Models, methods, and metrics," *Proc. IEEE*, vol. 102, no. 8, pp. 1283–1295, Aug. 2014.

[2] S. E. Quadir, *et al.*, "A survey on chip to system reverse engineering," *ACM J. Emerging Technol. Comput. Syst.*, vol. 13, no. 1, pp. 1–34, 2016.

[3] 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.

[4] A. Vijayakumar, V. C. Patil, D. E. Holcomb, C. Paar, and S. Kundu, "Physical design obfuscation of hardware: A comprehensive investigation of device and logic-level techniques," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 1, pp. 64–77, Jan. 2017.

[5] B. Shakya *et al.*, *Introduction to Hardware Obfuscation: Motivation, Methods and Evaluation*. Berlin, Germany: Springer, 2017, pp. 3–32.

[6] S. Wallat, M. Fyrbiak, M. Schlögel, and C. Paar, "A look at the dark side of hardware reverse engineering – A case study," in *Proc. IEEE 2nd Int. Verification Security Workshop*, 2017, pp. 95–100.

[7] 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.

[8] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Proc. 8th Int. Conf. Recent Advances Intrusion Detection*, 2005, pp. 207–226.

[9] X. Hu *et al.*, "Large-scale malware indexing using function-call graphs," in *Proc. 16th ACM Conf. Comput. Commun. Security*, 2009, pp. 611–620.

[10] M. Vujošević-Janičić *et al.*, "Software verification and graph similarity for automated evaluation of students' assignments," *Inf. Softw. Technol.*, vol. 55, no. 6, pp. 1004–1016, 2013.

[11] D. Koutra *et al.*, "Algorithms for graph similarity and subgraph matching," Tech. Rep. Carnegie-Mellon-University, 2011. [Online]. Available: https://people.eecs.berkeley.edu/~aramdas/reports/DBreport.pdf

[12] 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.

[13] U. Schöning, *Graph Isomorphism is in the Low Hierarchy*. Berlin, Germany: Springer, 1987, pp. 114–124.

[14] Y. Shi, C. W. Ting, B.-H. Gwee, and Y. Ren, "A highly efficient method for extracting FSMs from flattened gate-level netlist," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2010, pp. 2610–2613.

[15] W. V. Quine, "The problem of simplifying truth functions," *Amer. Math. Monthly*, vol. 59, no. 8, pp. 521–531, 1952.

[16] O. Sokolsky, S. Kannan, and I. Lee, "Simulation-based graph similarity," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2006, pp. 426–440.

[17] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Res. Logistics Quart.*, vol. 2, no. 1/2, pp. 83–97, Mar. 1955.

[18] P. P. F. Chan and C. Collberg, "A method to evaluate CFG comparison algorithms," in *Proc. 14th Int. Conf. Qual. Softw.*, 2014, pp. 95–104.

[19] A. E. Brouwer *et al.*, *Spectra of Graphs*. Berlin, Germany: Springer, 2012.

[20] B. Crawford, R. Gera, J. House, T. Knuth, and R. Miller, "Graph structure similarity using spectral graph theory," in *Proc. Int. Workshop Complex Netw. Appl.*, 2017, pp. 209–221.

[21] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proc. Int. Conf. World Wide Web*, 1998, pp. 107–117.

[22] T. U. Aoki Laboratory, *AES Encryption/Decryption Circuits With Different Sbox Implementation Strategies*, 2015. [Online]. Available: http://www.aoki.ecei.tohoku.ac.jp/crypto/web/cores.html

[23] N. S. Agency, *Rijndael128 Implementation*, 2000. [Online]. Available: http://csrc.nist.gov/archive/aes/round2/r2anlsys.htm#NSA

[24] P. Sasdrich and T. Güneysu, "A grain in the silicon: SCA-protected AES in less than 30 slices," in *Proc. IEEE 27th Int. Conf. Appl.-Specific Syst. Archit. Processors*, 2016, pp. 25–32.

[25] R. Herveille, *I2C Controller Core*, 2017. [Online]. Available: https://opencores.org/project,i2c

[26] M. J., *RISC5x Implementation*, 2011. [Online]. Available: https://opencores.org/project,risc5x

[27] O. Girard, *openMSP430 Implementation*, 2016. [Online]. Available: https://opencores.org/project,openmsp430

[28] H. Salmani, *Aes-T1000*, 2013. [Online]. Available: https://www.trust-hub.org/aes-t1000.php

[29] L. Li and H. Zhou, "Structural transformation for best-possible obfuscation of sequential circuits," in *Proc. IEEE Int. Symp. Hardware-Oriented Security Trust*, 2013, pp. 55–60.

[30] F. Rodríguez-Henríquez, N. A. Saqib, A. D. Perez, and C. K. Koc, *Cryptographic Algorithms on Reconfigurable Hardware*, 1st ed. Berlin, Germany: Springer, 2007.

[31] P. Swierczynski, M. Fyrbiak, P. Koppe, and C. Paar, "FPGA trojans through detecting and weakening of cryptographic primitives," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 8, pp. 1236–1249, Aug. 2015.

[32] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *Proc. IEEE 31st Int. Conf. Comput. Design*, 2013, pp. 471–474.

[33] 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.

[34] M. Fyrbiak *et al.*, "Hardware reverse engineering: Overview and open challenges," in *Proc. IEEE 2nd Int. Verification Security Workshop*, 2017, pp. 88–94.

[35] R. S. Chakraborty and S. Bhunia, "HARPOON: An obfuscation-based SoC design methodology for hardware protection," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 10, pp. 1493–1502, Oct. 2009.

[36] G. Li *et al.*, "Effective graph classification based on topological and label attributes," *Statist. Anal. Data Mining*, vol. 5, no. 4, pp. 265–283, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1002/sam.11153

[37] Z. Zhang, M. Wang, Y. Xiang, Y. Huang, and A. Nehorai, "RetGK: Graph kernels based on return probabilities of random walks," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 3968–3978.

[38] T. Meade, Y. Jin, M. Tehranipoor, and S. Zhang, "Gate-level netlist reverse engineering for hardware security: Control logic register identification," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2016, pp. 1334–1337.

[39] K. Zeng and P. M. Athanas, "A Q-Gram birthmarking approach to predicting reusable hardware," in *Proc. Design Autom. Test Europe Conf. Exhib.*, 2016, pp. 1112–1115.

[40] K. Hasegawa, M. Oya, M. Yanagisawa, and N. Togawa, "Hardware trojans classification for gate-level netlists based on machine learning," in *Proc. IEEE 22nd Int. Symp. On-Line Testing Robust Syst. Design*, 2016, pp. 203–206.

Marc Fyrbiak received the BSc degree in computer science from TU Braunschweig, Germany, in 2012, and the MSc degree in IT security from Ruhr-Universität Bochum, Germany, in 2014. He is currently working towards the PhD degree in the Chair for Embedded Security, under the supervision of C. Paar. His research interests include reverse engineering of hardware and embedded software systems.

Sebastian Wallat received the BSc degree in computer science from University Duisburg-Essen, Germany, in 2012, and the MSc degree in IT security from Ruhr-Universität Bochum, Germany, in 2016. He is currently working toward the PhD degree with the University of Massachusetts, Amherst, 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 hard- and software.

Sascha Reinhard received the BSc degree in IT security from Ruhr-Universität Bochum, Germany, in 2016. He is currently working toward the MSc degree in IT security with Ruhr-Universität Bochum, Germany.

Nicolai Bissantz received the PhD degree in science from the University of Basel, Switzerland, in 2001, and the habilitation degree in mathematics from the Ruhr-University Bochum, in 2015. He is currently director of studies with the Ruhr-University Bochum. His current research interests include statistical inverse problems and applied statistics in science and technology.

Christof Paar (Fellow, IEEE) received the MSc degree from the University of Siegen, and the PhD degree from the Institute of Experimental Mathematics, University of Essen, Germany. He holds the chair for embedded security with Ruhr-Universität Bochum, Bochum, Germany, and is an affiliated professor with the University of Massachusetts Amherst, Amherst, Massachusetts. His research interests include highly efficient software and hardware realizations of cryptography, physical security, security evaluation of real-world systems, and cryptanalytical hardware. He is a fellow of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.