RELIC-FUN: Logic Identification through Functional Signal Comparisons

James Geist*, Travis Meade*, Shaojie Zhang*, Yier Jin†
*Department of Computer Science, University of Central Florida
†Department of Electrical and Computer Engineering, University of Florida
jimg@knights.ucf.edu, travis.meade@ucf.edu, shzhang@cs.ucf.edu, yier.jin@ece.ufl.edu

Abstract—**The ability to reverse engineer a hardware netlist in order to detect malicious logic has become an important problem in recent years. Much work has been done on algorithmically identifying structure and state in circuits; the first step of which is to separate control signals from data signals. The most current tools rely on topological comparisons of logic in order to identify signals which are uniquely structured in the netlist, as these signals are likely control signals. However, topological comparisons become less effective when a netlist has been resynthesized and optimized. We present a new tool, RELIC-FUN, based on netlist slicing and functional comparison of logic. Experimental results show that depending on netlist size, optimization, and control logic density, the proposed algorithm can be more accurate, and faster, than existing topological algorithms in many cases.**

Keywords: Netlist Reverse Engineering, Logic Identification, Hardware Security

I. Introduction

Modern integrated circuits pass through many hands from initial design to release into the supply chain, and typically contain third party intellectual property (IP). Their specifications are written in a high level language such as Verilog or VHDL; these languages allow the engineer to specify the design at a high level of abstraction similar to a software programming language. As with software languages, many libraries are available to perform common functions in order to prevent engineers from having to repeatedly solve problems that arise over and over. These libraries, called IP cores, come from many sources, and are used despite their implementation being opaque to the circuit designer.

An overview of the generation process follows. A compiler translates (synthesis) design specifications into a network of logic gates. These gate descriptions are defined in a cell library specific to the targeted fabrication technology. The cells vary not just in the logical function but also in terms of timing, area, and power usage and are selected based on the optimization settings for design. The last stage of design places these cells into positions on the die, and interconnects them (place and route).

Often, a design will use not just individual cells but large pregenerated IP cores for which the engineer may not have source code. The only access the engineer has to the third party circuitry is the netlist after the design has been compiled and synthesized. Netlists may also be recovered from fabricated chips returned from the foundry. In either of these cases, malicious code may have been inserted into the design, either via extra logic in a third party IP core, or changes to the design by a malicious foundry. Again access to the original source code might be limited, and even with access to the high level specification, as the netlist is at gate level, it cannot just be compared to the original source code. Verification that the design functions only as intended is difficult.

One solution to this problem is to use reverse engineering tools on the final netlist to discover pieces of the original design, and then simulate the circuit to ensure that unintended states cannot be reached. In most stateful designs, a *Finite State Machine* (FSM) is used to generate control signals that synchronize the functional units in the rest of the design. An FSM can be represented by a set of registers that change on every clock cycle based on the state of the design's inputs and functional units. Reverse engineering tools allow the designer to isolate the FSM's in the design, and then to simulate them looking for atypical state combinations that are the result of malware injection and/or hardware faults. Additional tools might be required, if the intended states are unknown.

This paper makes the following contributions:

1) A logic identification algorithm which uses functional rather than topological matching.
2) A tool, RELIC-FUN, based on a functional matching algorithm which is more accurate in many cases than topological algorithms.
3) A comparison of how RELIC-FUN and an existing topological tool, RELIC, perform on both unoptimized and optimized netlists.

The rest of the paper is organized as follows. Section II discusses the motivation for the algorithm application we present, and section III covers related work in the field. Section IV provides formal definitions for the theory behind the algorithm, and section V details the algorithm itself. Section VI describes our experimental setup, and section VII discusses comparative results between our algorithm and the existing tool on sample netlists. Finally, section VIII summarizes the work.

II. Motivation

One of the primary uses of reverse engineering is to determine if malicious elements (malware) have been introduced into a design at some point in the fabrication process. A common implementation of malware is for the corrupted design to react to an unlikely or invalid combination of inputs. In the original design, the hardware might report an error in such a case. The

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

malware, however, recognizes a particular invalid input combination as a signal for the hardware to enter an unanticipated but functional state. Once in that state, the hardware may execute any number of undesirable actions, such as leak data through a side channel back to the attacker. Implementations of this type are typically centered around extensions to an FSM responsible for generating control signals.

The definition of an FSM is a set of states with transitions between them; the FSM is in exactly one state at any given time. On each clock cycle, the FSM moves into a new state based on its input signals. The input signals are typically input pins or state signals from functional units. Each state corresponds to setting the output control signals of the FSM to certain values. If there are n control signals, then there are 2^n possible states; however, in most FSM's only a subset of the states are valid. If malicious code has been inserted, then the FSM will have extra states that the designer did not intend.

In order to detect FSM tampering, a reverse engineer must first find the FSM in the netlist. Very little of the original high level design may be known other than the input and output connections. Logic identification tools are used to isolate control signals; such control signals are often the output of registers which are part of an FSM. Once a part of the FSM has been found, the rest can be pieced together by the tight-knit connections between its registers. Then a simulation program such as REFSM [8] can be used to explore the output space of the FSM to search for atypical transitions.

When a netlist is compiled, the compiler can choose to perform varying levels of optimizations. Optimizations are an attempt to improve some metrics of the final implementation, such as lowering the number of gates in order to make the die smaller or power consumption lower. Consider the case where we have a multi-bit adder made up of one-bit adders. The carry out of each component adder is connected to the carry in of the next bit. At the low bit end, the carry in will be connected to some other logic, such as a CPU carry flag register. Likewise the carry out of the final bit will be connected to some other logic (perhaps the input to the same carry flag register). The compiler may decide that, for either of these cases, that it is more efficient to combine part of the adder with the logic that drives the low bit carry on or uses the high bit carry out. In general, optimization can cause a blurring of the line between functional units by creating gates that function as part of multiple functional units.

Tools which use topological comparison can be misled by optimization; in the previous example, carry in is clearly data, but because optimization has changed the carry in logic in the low bit adder, that signal is now different topologically from all of the other carry in's and may be mis-identified as a control signal. These false positives are noise in the output of topological matching tools, making it more difficult to discern real control signals. We introduce a new tool, RELIC-FUN, which is designed to overcome these difficulties by performing functional rather than topological matching.

III. Related Works

Multiple algorithms for partitioning a netlist into control and data signals have been proposed. They can be classified into two approaches which are complementary: isolate control signals, or aggregate data path signals. Further, they can be classified by whether they test the netlist functionally or topologically. Functional algorithms attempt to match parts of the netlist based on the computed Boolean function; topological algorithms match based on similar gate structures.

	Topological	Functional
Data Path	REBUS [11] REPCA [11]	WordRev [6]
Control Logic	RELIC [7] fastRELIC [12]	**RELIC-FUN** (This Work)

TABLE I: Classification of Algorithms.

Across all algorithms, the common theme is to use the inherent replication of structure that naturally occurs in the implementation of any multi-bit data path to separate data signals from control signals.

RELIC [7] performs a similarity test between all pairs of register inputs as part of a larger feature set used for an unsupervised learning task to assign likelihood scores that a signal is a control signal. Given two register inputs, the algorithm computes a similarity score in the range $[0..1]$ based on how structurally similar the inputs' fanin cones are. RELIC compares every candidate pair of signals in the netlist, and generate a Pairwise Similarity Score (PSS) for each pair. Thresholding is used to cull the list of candidate pairs down in two places: on the original similarity score described above, and on the final PSS generated by a normalized bipartite maximal matching of all remaining pairs.

fastRELIC [12] was proposed as a performance enhancement over RELIC. It is functionally very similar to RELIC; the main difference is that it considers the inversion of signal pairs (i.e. if comparing A and B, then A and $\overline{B}$ are also compared) as also proposed by Meade et al. as an improvement to RELIC itself [11]. fastRELIC improves on performance by grouping signals based on their similarity scores as soon as the original comparison is done. By making inferences about score relationships while building the groups, far less comparisons between signal pairs need to be performed. As the comparison is a recursive search through the netlist, this results in substantial performance gain. Unfortunately, as the authors chose to reimplement RELIC in Python and then make their improvements on that implementation, a direct performance comparison between fastRELIC and RELIC-FUN (which, like RELIC, is implemented in C++) is difficult.

WordRev algorithm [3] partitions the netlist into equivalence classes of k-feasible cuts such that all members of each class compute the same Boolean function. The algorithm proceeds by attempting to find connections between members of a class; for example, a chain of 1-bit adders forming a multi-bit adder, which in turn is part of a data path. Pieces of the data path are connected via *forward propagation*; i.e. searching from a word's register outputs to nearby signals to find the next word in the path.

RELIC-FUN shares some algorithmic heritage with WordRev [6]. Both algorithms cut and functionally compare pieces of the netlist in similar ways. However, WordRev is interested in finding similar parts of the netlist in order to determine if they are part of a larger functional structure such as a multi-bit word; RELIC-FUN is interested in finding pieces of the netlist

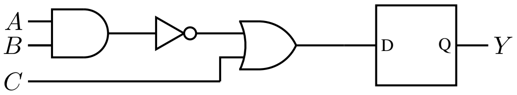

Fig. 1: A 3-feasible slice

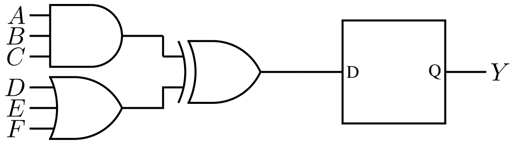

Fig. 2: A circuit with multiple possible slices

which are *not* similar to many other pieces, as these are more likely to be control signals.

REBUS and REPCA are two tools proposed by Meade et al. [11]. REBUS combines the similarity score of RELIC with the forward propagation data path discovery of WordRev to reconstruct the data paths of a netlist. REPCA runs Principal Component Analysis (PCA) on various topological features such as fanin size, fanout size, and number of registers to input or output. The top axes returned by PCA are then used to build clusters of similar signals.

IV. Definitions

In this section, we will introduce all notations/definitions which will be used in our algorithm. An *equivalence class* of a set S is a subset $E \subseteq S$ such that all members of E are considered equal by some equivalence relation. A *partitioning* of S places every member of S into exactly one equivalence class; i.e. if EQ is the set of all equivalence classes in a partitioning of S, then $\cup_{E \in EQ} E = S$ and $\cap_{E \in EQ} = \varnothing$.

A *slice* is a subset of the netlist, defined for a given signal Y which is the slice's output, with a set of captured input signals $A_1, A_2, ..., A_n$ selected from the transitive fanin of Y. The slice computes a Boolean function $Y = f(A_1, A_2, ..., A_n)$; the inputs must totally determine Y. Meeting this condition requires that no gate contained in the slice has uncaptured, non-constant inputs. WordRev [3] refers to a slice of this form as a *feasible cut* of the output Y, and calls a conforming slice of k inputs *k-feasible*.

Figure 1 shows a simple 3-feasible slice. If the AND gate had one more input, the slicing algorithm would have to include it, in order to satisfy the requirement that all inputs of included gates must be captured. The algorithm, in this case, could also not just omit C, as that would leave the OR gate with one input dangling.

Figure 2 shows a more complex circuit. A 2-feasible cut of this example is trivial; we take the two inputs of the XOR gate. A 3-feasible cut, however, is impossible. There are two 4-feasible cuts: A, B, C, and the second input to the XOR gate, or D, E, F, and the first input of the XOR gate. A 5-feasible cut is again impossible, and a 6-feasible uses all of A, B, C, D, E, F. In general, for any given k, there can be zero or more k-feasible cuts on a given signal.

We say that two slices are *functionally equal* if they both compute the same Boolean function on the same set of inputs, potentially permuted in different ways. Assume we have two circuits with outputs X and Y respectively. Both circuits take the same inputs $A_1, A_2, ..., A_n$. Further, assume that $X = f(A_1, A_2, ..., A_n)$ is the Boolean function computed by X's fanin. We consider X and Y to be functionally equivalent if, for some permutation π of the inputs, $Y = f(\pi(A_1, A_2, ..., A_n))$. In Table II, X and Y have three inputs each. Their outputs differ for the order of inputs given; however, the truth tables are equal. If we permute the inputs such that ABC is given to circuit X and CBA is given to circuit Y, then the outputs X and Y match for all inputs.

A	B	C	X	Y
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	1
1	1	1	1	1

TABLE II: Two functionally equivalent truth tables

V. RELIC-FUN

A. *RELIC-FUN Algorithm*

Algorithm 1 Compute equivalence classes

```
function relic-fun($signals, target - size$)
    $classes \leftarrow []$
    for $s \in signals$ do
        $sl \leftarrow slice(s, target - size)$
        $found \leftarrow false$
        for $cl \in classes$ do
            $t \leftarrow cl.first$
            if $ttequal(sl, t)$ and not $found$ then
                $add(cl, sl)$
                $found \leftarrow true$
            end if
        end for
        if not $found$ then
            $newset \leftarrow set(sl)$
            $add(classes, newset)$
        end if
    end for
    $sort(classes, \lambda x : x.size)$
```

Algorithm 1 shows the RELIC-FUN algorithm for classifying signals. The input $signals$ is the set of interesting signals to classify; RELIC-FUN takes this as all signals that are register inputs. For each signal, the algorithm attempts to find a k-feasible cut for the signal by executing a breadth-first search (BFS) backwards from the signal into its fanin logic, constructing a set I of all gate inputs not totally interior to the slice. The search terminates once I contains k or more elements. As we showed in Figure 2, there is no guarantee that a k-feasible cut exists for a given signal. If there is no k-feasible cut for a signal, RELIC-FUN will keep the next larger slice it can find.

For efficiency, RELIC-FUN stores truth tables as 64 bit integers; each bit is the result of assignment of one set of Boolean values to the inputs of the slice. This imposes a practical limit of $k \leq 6$ since $2^6 = 64$. If there is no slice with a number of inputs less than this implementation defined maximum, then the signal will be ignored. In practice, this does not cause problems, as good results can be obtained with k of 4 and a case where slices are discarded because of this limit is extremely unlikely with that target k.

If there are multiple slices for a given k, then RELIC-FUN keeps only the first slice encountered. In practice this also produces good results, as the slices are all searched with the same BFS algorithm and thus the gates are enumerated in the same order. However, this implementation detail introduces a dependence on topology; future work will investigate examining all possible k-feasible cuts of a signal and intelligently combining the results if there are multiple cuts.

Each new slice is then compared to a candidate slice from the equivalence classes built so far, using the definition of functional equality. By definition, all members of an equivalence class are equivalent, so only one comparison per class is needed. If the slice matches an existing class, then it is added to that class; otherwise, a new class is created with the new slice as its only member. As mentioned above, the slicing algorithm may generate slices with varying number of inputs; each equivalence class can only contain slices with the same number of inputs.

Algorithm 2 Test functional equality

```
function ttequals($A$, $B$)
    if $|A.inputs| \neq |B.inputs|$ then
        return $false$
    end if
    for $p \in permutations(|A.inputs|)$ do
        $match \leftarrow true$
        for $i \in combinations(\{0, 1\}, |A.inputs|)$ do
            if $A(i) \neq B(p(i))$ then
                $match \leftarrow false$
            end if
        end for
        if $match$ then
            return $true$
        end if
    end for
    return $false$
```

The functional equivalence algorithm `ttequals` is shown in algorithm 2. Two slices A and B are compared by brute-force enumeration of all possible input permutations. If A and B have different inputs sizes, then they are not equal by default. If the input sizes are the same, then `ttequals` generates all permutations of a set of inputs of that size. For each permutation, A and B are computed for each possible set of input values; B is given the current permutation's reordering of the inputs given to A. If, for any permutation, A and B match on all possible inputs, then A and B are functionally equivalent. This problem has been well-studied as it occurs often in VLSI compilers (see, for example, Mohnke and Malik [1]); however, we did not implement any heuristics as the input sizes RELIC-FUN works with are so small that performance is not an issue.

After the equivalence classes are built, they are sorted by size. Classes with few members are likely to be control logic as they compute functions dissimilar to the rest of the design. Classes with a large number of members, on the other hand, are likely to contain data registers, as multi-bit register structures are very self-similar. In practice, only examining signals from singleton classes yields good results.

B. *Runtime Analysis*

The function `ttequals`, which determines if two slices are functionally equal, is computationally non-trivial as we do not know the order of the inputs to each slice. Therefore the algorithm must compare each permutation of inputs of one slice to the other slice; for each permutation, the entire truth table must be evaluated. If there are k inputs, this is $O(k!2^k)$. The algorithm is tenable because it does not need to compare the truth tables of every pair of signals; it only needs to compare each signal against one signal from each equivalence class. This is the key observation that allows the algorithm to run in reasonable time. If there are n signals and c classes, the total runtime is $O(k!2^k cn)$. k is a parameter and is typically very small (4 to 6). On practical netlists c also tends to be small (around 100 or less).

VI. Experimental Setup

We implemented RELIC-FUN in C++ and compared it against RELIC, which is also written in C++. We ran both programs against 7 netlists chosen based on size and balance between logic and data. The target netlists were compiled using Design Compiler [15] and synthesized using the Nangate OpenCell 15nm library. The signals investigated were from the smallest one or two equivalence classes returned by RELIC-FUN, and above a gap in the Z-score values returned by RELIC. Each signal was classified into one of three bins:

1) High quality if the signal could be seen to be part of an FSM, interconnecting two parts of the design, or otherwise implemented a standalone, high level function like reset.
2) Medium quality if the signal was clearly not data, but was glue logic or control data (for example, a counter for UART timing).
3) Low quality if the signal could definitively be seen to be part of a data path.

Two versions of each netlist were evaluated. The original version compiled by Design Compiler is representative of a design as it is being developed and simulated. Reverse engineering these netlists is intended to discover malware embedded in third party IP cores. The second version is the original version, resynthesized with further optimizations using the Berkeley ABC toolset [13]. This version is representative of a design after manufacture. Such netlists may, in practice, be recovered from actual hardware by a process of chemically exposing the die inside its packaging, using optical or electron microscopy to capture the die's structure, and algorithmically processing the images back into the represented gates [2]. This process is intended to catch malicious modifications introduced during fabrication.

All designs except for AES Trojan are from OpenCores [16]. The AES Trojan is per Trust-Hub [5], [10]. All timing runs were executed on an Intel(R) Core(TM) i5-9600K system running at 3.70GHz with 32G of memory.

VII. Experimental Results

A. *Accuracy*

Table III shows the results against the compiled test netlists; table IV shows the results on the same netlists after having

			RELIC				RELIC-FUN			
Netlist	Class	Gates	**H%**	M%	L%	#	**H%**	M%	L%	#
UART	Logic	168	**28**	45	28	29	**33**	50	17	24
DES	Pure Data	1984	**0**	0	100	3	**0**	0	0	0
RSA	Data	2139	**67**	0	33	3	**27**	27	45	11
ETH	Logic	2653	**25**	0	75	8	**56**	35	8	48
MC8051	Balanced	6590	**63**	19	19	43	**47**	30	23	43
AES	Data	9114	**0**	100	0	3	**14**	86	0	14
AES Trojan	Pure Data	11175	**0**	100	0	9	**0**	50	50	2

TABLE III: Results on Original Netlists described using % High (which are emboldened), % Medium, % Low Quality Signals; Number of Signals Considered. Winners underlined.

			RELIC				RELIC-FUN			
Netlist	Class	Gates	**H%**	M%	L%	#	**H%**	M%	L%	#
UART	Logic	168	**50**	50	0	8	**50**	33	17	6
DES	Pure Data	1984	**0**	0	0	0	**0**	0	0	0
RSA	Data	2139	**0**	0	100	4	**100**	0	0	3
ETH	Logic	2653	**60**	0	40	5	**65**	26	9	23
MC8051	Balanced	6590	**57**	43	0	7	**100**	0	0	6
AES	Data	9114	**11**	89	0	9	**11**	89	0	9
AES Trojan	Pure Data	11175	**0**	100	0	5	**0**	50	50	2

TABLE IV: Results on Resynthesized Netlists described using % High (which are emboldened), % Medium, % Low Quality Signals; Number of Signals Considered. Winners underlined.

been resynthesized with ABC. We compare accuracy based on the percentage of high quality signals each algorithm reveals.

In the original designs that have not been heavily optimized, RELIC beats RELIC-FUN in two cases. On the resynthesized design, however, RELIC-FUN performs better. It ties or exceeds RELIC in every cases.

The UART core is the smallest test case with only 168 gates. It contains a transmitter and receiver which are largely separate components. The data path is a small percentage of the design: two shift registers, one for the receiver and one for the transmitter. The rest of the design is logic, consisting of counters to divide the clock to the correct rate to shift the serial data in and out and to count the number of bits transmitted or received. The transmitter and receiver both have an FSM. We considered any signal that was part of one of the two FSM's as high quality; any signal that was part of a counter as medium quality; and signals which were part of the shift or I/O data registers as low quality. RELIC and RELIC-FUN performed similarly on this design, both in the number of signals recovered and the percentage of high quality signals. On the original netlist, RELIC-FUN works slightly better (33% versus 28% high quality signals recovered) and on the resynthesized netlist, both algorithms find 50% high quality signals. RELIC-FUN performs slightly worse on this case, finding one low quality signal, while RELIC finds all high and medium quality signals.

The DES core we included is a special case as it is purely pipelined and has no control logic. This design was included to test for false positives. RELIC-FUN properly finds no control signals at all – there are no small equivalence classes in the results. Original RELIC finds some false positives in the original version, but not in the resynthesized version. On the resynthesized version, both algorithms properly find no control signals.

The RSA core, like the DES core, is data heavy; however, while the DES algorithm only includes operations such as signal permutation and XOR which can be implemented in combinational logic [14], RSA requires multiplication [9]. Thus the RSA core contains a multiplication functional unit and associated control signals. On the original design, RELIC-FUN finds a lower percentage of high quality signals; however, after resynthesis, RELIC loses track of any high quality signals (finding false positives in the multiplier data path), while RELIC-FUN only finds high quality signals.

The ETH core is an Ethernet controller. This core is logic heavy, for the same reasons as the UART; however, as Ethernet is a more complex protocol with framing and addressing, it is much larger. RELIC-FUN fares better than RELIC on both the original and resynthsized design in this case, finding a large number of control and FSM signals. RELIC appears to have problems distinguishing between control signals and data paths which are not totally consistent across bits; in this case, it tags a number of signals containing the cyclic redundancy check (CRC) of payload data as control.

The MC8051 is a popular, open source, 8 bit microcontroller [4]. The fact that it is open source lends it to manual classification of signals. The core has a large amount of both control logic and data (registers, ALU, etc.) and thus we considered it a balanced design. It also contains a varying number of serial interface and timer/counter units; the implementation we studied has one of each. The design optimizes well upon resynthesis, and RELIC-FUN shows clear advantage on that version of the netlist.

We studied two AES implmentations: an AES with a round counter, and an AES core with an inserted hardware Trojan. The AES core is data heavy and the Trojan'ed version is a streaming implementation which has no control logic (the goal in this design is to isolate the Trojan.) We classify AES as data heavy and AES Trojan as pure data. Both algorithms tie for both cases of these cores, except for the the unoptimized AES, where RELIC-FUN wins. Both algorithms discovered signals from the Trojan on both the original and optimized netlists.

Overall in the resynthsized test cases, RELIC-FUN either wins or ties. On the DES core, there is no control logic to be found, so a tie of zero signals is to be expected. On AES, there are only two high quality signals: one to load new data, and one to signal that the encryption is complete. In AES Trojan, as in

DES, the implementation has no control signals. This leaves 4 netlists with a non-trivial number of control signals to discover: UART, RSA, ETH, and MC8051. REFLIC-FUN outperforms RELIC in all of these except UART, where the two algorithms tie.

B. Experimental Runtime

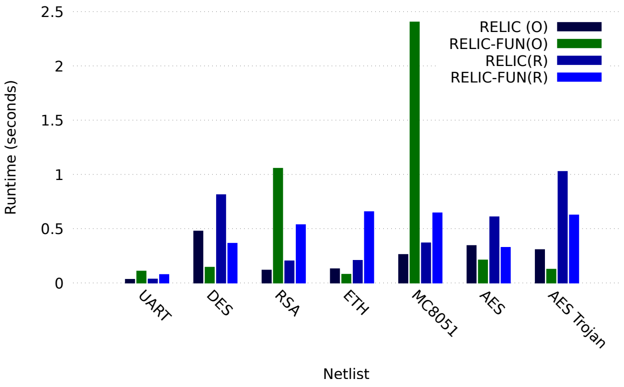

Fig. 3: Runtimes on Original (O) and Resynthsized (R) Netlists

Figure 3 compares the runtimes of RELIC versus RELIC-FUN on each test netlist. Each timing is the average of ten runs of the command. On many of the original netlists, RELIC-FUN is faster than RELIC; a notable exception is MC8051, where it is almost an order of magnitude slower. RELIC's performance is mostly dominated by the overall size of the netlist. RELIC-FUN's, as shown in the analysis above, is a combination of the netlist size and the number of classes discovered. The MC8051 is large in both ways: RELIC-FUN found 116 equivalence classes, and it has 6500 gates. The two AES cases have more gates, but are very symmetric and thus only have a small number of classes (20 for the AES, 9 for the AES Trojan).

C. Discussion

We compared RELIC-FUN in depth with one other popular tool, RELIC, and considered two main scenarios. The first, the unoptimized netlist, is useful when a design is being compiled during development. In this case, reverse engineering is likely looking for malware injected via third party IP core. RELIC has advantages here, as the topology has not yet been corrupted. The second case, the optimized netlist, is more likely to occur when the netlist has been recovered directly from hardware. In this case, RELIC-FUN outperforms RELIC and is the more likely candidate for analysis. On the resynthsized netlists, the only case where RELIC-FUN becomes slower is ETH. In all other cases the relationship between the two tools is the same as for the original netlists.

On large designs which generate many equivalence classes, RELIC-FUN can be slower than RELIC. In the netlists we tested, this can be seen in the unoptimized Ethernet design. There may be cases where RELIC is a better choice if the runtime of RELIC-FUN is too large. The performance of RELIC-FUN, however, is much more comparable to RELIC on optimized designs, were RELIC is more accurate as well. Future work is planned to improve the performance of RELIC-FUN in all cases.

VIII. Conclusion

In this paper we have proposed a functional algorithm for logic identification that can outperform topological algorithms in optimized circuits. RELIC-FUN ties or exceeds the percentage of high quality signals recovered compared to RELIC on the optimized versions of all netlists we studied, and in many cases had lower runtime than RELIC. There are many improvements to RELIC-FUN (such as heuristics on functional equality to improve performance, and examining all possible slices of a signal to improve accuracy) that we believe can make RELIC-FUN even more competitive compared to other logic identification algorithms.

Acknowledgements

This work is partially supported by the National Science Foundation (NSF-1812071) and DARPA.

References

[1] J. Mohnke and S. Malik, "Permutation and phase independent boolean comparison," in *1993 European Conference on Design Automation with the European Event in ASIC Design*, IEEE, 1993, pp 86–92.

[2] R. Torrance and R. James, "The state-of-the-art in semiconductor reverse engineering," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, IEEE, 2011.

[3] W. Li, A. Gascon, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia, "Wordrev: Finding word-level structures in a sea of bit-level gates," in *2013 IEEE international symposium on hardware-oriented security and trust (HOST)*, IEEE, 2013, pp 67–74.

[4] *Mc8051 ip core synthesizeable vhdl microcontroller ip-core user guide*, version 1.2, Oregano Systems, 2013. [Online]. Available: https://www.oreganosystems.at/application/files/9815/3313/6275/mc8051_ug.pdf.

[5] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmark development," in *IEEE Int. Conference on Computer Design (ICCD)*, IEEE, 2013.

[6] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, "Reverse engineering digital circuits using functional analysis," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, IEEE, 2013.

[7] T. Meade, Y. Jin, M. Tehranipoor, and S. Zhang, "Gate-level netlist reverse engineering for hardware security: Control logic register identification," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2016.

[8] T. Meade, S. Zhang, and Y. Jin, "Netlist reverse engineering for high-level functionality reconstruction," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, 2016.

[9] K. Moriarty, E. Corporation, B. Kaliski, Verisign, J. Johnsson, S. AB, A. Rusch, and RSA, "Pkcs 1: Rsa cryptography specifications version 2.2," Moriarty, K., Tech. Rep. 8017, Nov. 2016, pp 1–78. [Online]. Available: https://tools.ietf.org/html/rfc8017.

[10] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, and M. Tehranipoor, "Benchmarking of hardware trojans and maliciously affected circuits," *Journal of Hardware and Systems Security (HaSS)*, Apr. 2017.

[11] T. Meade, K. Shamsi, T. Le, J. Di, S. Z. Zhang, and Y. Jin, "The old frontier of reverse engineering: Netlist partitioning," *Journal of Hardware and Systems Security*, vol. 2, no. 3, pp 201–213, 2018.

[12] M. Brunner, J. Baehr, and G. Sigl, "Improving on state register identification in sequential hardware reverse engineering," in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, Washington, D.C., USA, May 2019, pp. 151–160. DOI: 10.1109/HST.2019.8740844.

[13] *Abc: A system for sequential synthesis and verification*, Berkeley Logic Synthesis and Verification Group. [Online]. Available: https://people.eecs.berkeley.edu/~alanmi/abc/.

[14] *Data encryption standard (des)*. [Online]. Available: https://csrc.nist.gov/CSRC/media/Publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf.

[15] *Design compiler*. [Online]. Available: https://www.synopsys.com/.

[16] *Opencores.org*. [Online]. Available: https://opencores.org/.