On Using Control Signals for Word-Level Identification in A Gate-Level Netlist

Edward Tashjian, Azadeh Davoodi
Department of Electrical and Computer Engineering
University of Wisconsin at Madison, USA
{etashjian, adavoodi}@wisc.edu

ABSTRACT

This work tackles the problem of reverse engineering a gate-level netlist in order to identify groups of wires corresponding to words. It serves as the major step to find high-level modules and analyze their correct functionality in the presence of Hardware Trojans. Our core idea is to find and utilize control signals to more effectively identify words. Specifically, modern designs provide ample opportunities because they contain numerous control signals which are automatically inserted by the CAD tools. But finding control signals is itself an unresolved challenge. We propose a procedure to identify words which at its core finds and utilizes a small subset of relevant control signals by exploiting partial structural similarity. In our experiments, we show the effectiveness of our procedure by showing a high number of identified words with high accuracy using many benchmarks with already-identified words as the reference case.

Categories & Subject Descriptors K.6.5. [Management of Computing and Information Systems]: Security and Protection

General Terms Security, Design

Keywords Control Signal Identification, Structural Matching, Reverse Engineering

1. INTRODUCTION

Hardware Trojans are a serious concern as systems are increasingly composed using off-the-shelf components. Even a few lines of alteration in the Hardware Description Language (HDL) of an Intellectual Property core that is purchased from a third-party vendor (for example to build a System on a Chip) is enough to introduce malicious behavior [14]. Hardware Trojans introduced at this level of abstraction can have a small area footprint and be easily obfuscated in the layout as they go through synthesis and optimization with the rest of the design [5]. Moreover, they may be triggered under rare events which may not be detected even using exhaustive test and simulation [10].

DAC '15, June 7 - 11, 2015, San Francisco, California, USA.

http://dx.doi.org/10.1145/2744769.2744878.

On-going research aims to detect Hardware Trojans at the register transfer level [3, 13]. But, Hardware Trojans may also be inserted during the synthesis and optimization steps to create a gate-level netlist. This may be done by a malicious designer and/or a malicious CAD tool [8, 11]. Finding Hardware Trojans in gate-level netlists is particularly challenging since the netlist may have been flattened thereby any trace of the design hierarchy is removed. Various optimization steps may also result in significant gate-sharing among various components.

For the challenging scenario described above, a major step towards identifying Hardware Trojans is to reverse engineer the gate-level netlist in order to identify high-level components [6, 9]. The identified high-level components can then be inspected to verify correct functionality or reconstruct an HDL description of the design. This reverse engineering problem becomes most challenging when there isn't knowledge of the potentially-used design patterns, library templates, and high-level design specifications to facilitate the process.

The first and major step towards implementing this detection process for the most-challenging variation described above, requires grouping of wires corresponding to individual words. The identified words can then be used to more easily find high-level components since inputs and outputs of the high-level components are often connected to one or more words. For example consider a simple addition of two 32-bit data words to create a new 32-bit data word as a line in an HDL program. In a gate-level netlist, the computational unit responsible for the addition can be more easily identified, if first, the three 32-bit wires corresponding to the two inputs and output words are identified.

Prior work on identifying words in a gate-level netlist comprises of applying structural and functional techniques [6, 7, 9]. Some of the structural techniques are limited to discovery against potentially-used design patterns [7] but the recent work [6, 9] addresses this limitation and proposes techniques which do not require any such knowledge. The functional techniques usually require some structural processing such as finding and enumerating cuts of certain size in the gate-level netlist [6]. So they may be applied after words are identified using a structural technique to further improve the word identification process.

In this work, we also study word identification in a gate-level netlist for the most challenging version of the reverse engineering problem in which there is no information about potentially-used library templates and high-level design specifications.

Our key observation is that there are cases in which the circuity implementing bits of a word may be partially-similar in structure. The dissimilar portions often involve control signal circuitry. Our main focus is to discover and utilize these control signals such that the "reduced circuitry", after assigning feasible values to control signals, is more likely comprised of similar structures. This allows new words to be discovered. State-of-the-art procedures miss such cases and require fully-matched circuit structures to discover individual bits in a word.

To identify control signals, an early work on reverse engineering the ISCAS-85 combinational circuits [2] provides strategies which are manually applied. However identification of control signals in modern design is itself a major challenge and a relatively new and unresolved problem. This is because modern designs now contain numerous control signals which are automatically inserted by CAD tools anywhere in the netlist and throughout the design flow. Examples include signals inserted to select scan mode, clock and power gating enable signals, etc. The designer may not have much knowledge about the locations of the control signals beyond the primary inputs since the majority may be inserted by the CAD tools as internal signals. A recent work from Synopsys [4] studies techniques for finding a category of "top-level" control signals of (up to a few thousand) to facilitate IC verification. However, the techniques are time-consuming to be integrated with the already-complex reverse engineering problem. Moreover, the identified control signals in [4] are limited to the category of top-level signals.

Our Contributions: In this work we propose a procedure to identify a small subset of relevant control signals which is integrated within the word-identification procedure. A novelty of our work is that we are not targeting identifying all the control signals. Rather, our procedure integrates identification of relevant control signals together with their suitable and feasible value assignments for the purpose of improving the word identification process.

We define relevant control signals such that the reduced circuitry after applying the assigned values to these control signals, is more likely to comprise of similar structures which implement bits within individual words. This allows identifying a higher number of words as well as more accurately identifying the individual words via having a lower fragmentation rate for the partially-matched words. To identify useful control signals, our integrated approach utilizes both structural similarities and structural *dis*similarities.

Our techniques are compatible and can be integrated with other works on word identification and reverse engineering such as [6, 7, 9] by providing them the reduced circuits, after utilizing the control signals to further identify words.

In our experiments, we report different metrics for evaluating the accuracy of the identified words in addition to the ones explored in the state-of-the-art. Another improvement compared to prior work is that in our experiments we use a setup that for each standard benchmark we can manually identify a set of actual words. This allows creating a golden reference case and experimenting with a high number of benchmarks for a more reliable validation.

The rest of the paper is organized as follows. Our procedure is explained in Section 2. It includes an overview in Section 2.1 with the details presented in Sections 2.2 to 2.6. Simulation results are presented in Section 3 and conclusions are offered in Section 4.

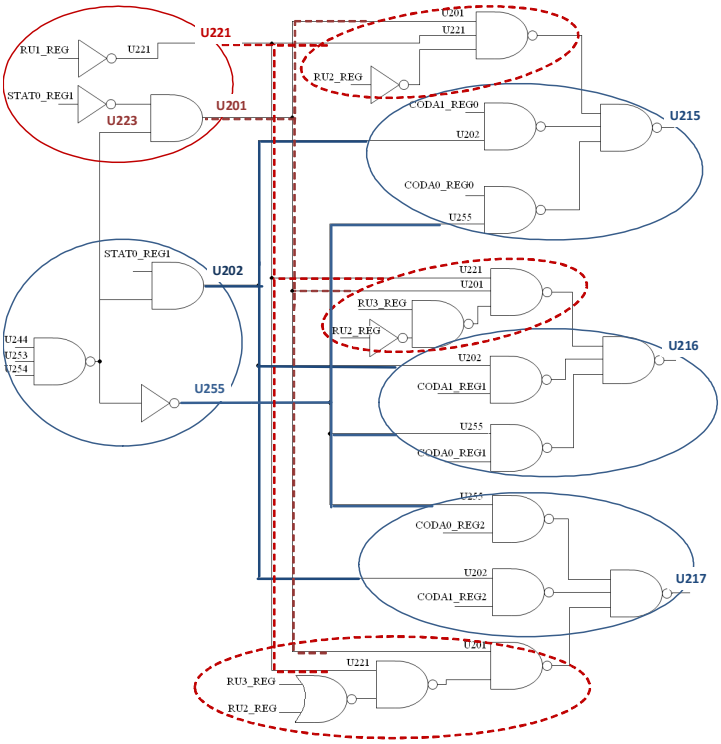

Figure 1: Fanin-cones of a 3-bit word in benchmark `b03`. Each bit has two subtrees of similar structure (shown in blue circles) and a dissimilar one (shown in dashed red circle) which are fed by control signals `U201` and `U221`.

2. OUR TECHNIQUES

We first give an overall view of our framework using an example. We then discuss various steps of our procedure in detail.

2.1 Overview

Figure 1 shows a gate-level implementation of a 3-bit word in benchmark `b03` which is an arbiter circuit from the ITC99 suite [1]. The bits are `U215, U216, U217` which were identified from the RTL code[1].

For the circuitry implementing each bit we show its fanin-cone down to four levels of logic gates. Because of various CAD tool optimizations and netlist flattening, it is unlikely that the logic levels beyond this will have any similarity in structure; a recent word detection technique is based on having similar fanin-cones between 2 to 4 levels [6].

For each bit, the fanin-cone has two subtrees with similar structure (circled in blue) as well as a dissimilar subtree (circled in dashed red). These together create 3 possibilities that a bit can take which are selected by control signals.

The similar subtrees share part of their fanin cone which generates two common control signals `U202` and `U255`. They select between `CODA0_REGi` to `CODA1_REGi`, for each bit i (from 0 to 2). The dissimilar subtrees are fed by common control signals `U221` and `U201` which are implemented by the

[1]Both gate-level and RTL implementations were downloaded from http://www.cad.polito.it/downloads/tools/itc99.html

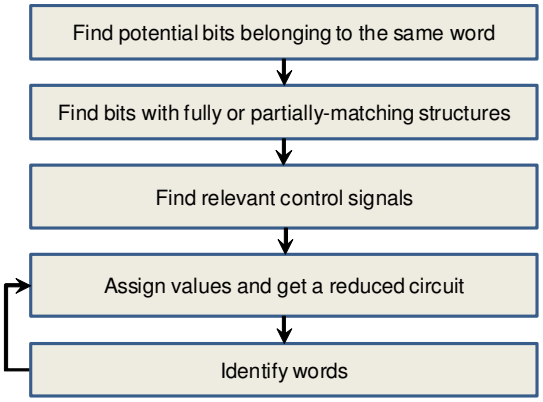

Figure 2: Flow chart of our techniques.

common fanin cone shown in the red circle. These two common control signals are combined in different ways (shown in the dashed circles) to create one of the three options that each bit can take which is selecting `RU2_REG` or `RU3_REG`.

Existing techniques for word identification cannot detect this 3-bit word because the fanin cones of the bits are not fully similar—either structurally or functionally. In our proposed technique, we address this limitation by allowing partial matching of subtrees.

Our approach first identifies bits which may potentially belong to the same word. For this example, these are `U215, U216, U217`. It then detects the two similar subtrees among the fanin-cones as well as the dissimilar subtrees. Among the dissimilar subtrees, it tries to find the common portion which in this case is the fanin-cone generating the common control signals `U221` and `U201`. It then searches for a suitable signal assignment for `U221` and `U201` which will be 0 in this case since they both feed NAND gates. It then simplifies the fanin-cones based on the signal assignments which removes the red dashed areas, leaving behind the blue circles as fanin-cones of the 3-bit word. These fanin-cones will now be fully similar in the simplified circuit. Therefore the 3 bits can be identified to belong to the same word.

At this stage, word discovery can be easily done using our technique since the similar structures were already identified. However, the simplified circuit can also be fed as input to existing structural or functional word-identification techniques so our work can be integrated with and be useful for other works on reverse engineering.

Figure 2 shows the generic flowchart describing our techniques. We first identify potential bits that may belong to the same word. Next a quick technique is used to find bits with fully or partially -matching structures within the bit-groupings found from the previous step. This step also stores the dissimilar portions for partially-matching structures. Relevant control signals are then identified among the dissimilar portions. Next, a set of suitable and feasible values are assigned to the control signals and the circuit is simplified based on the assigned values. For the simplified circuit, we then check if bits can be grouped together into words. If no words can be found, a different value assignment will be made to the control signals. The process continues until a limited set of possibilities have been explored. Next, we discuss the details of the above steps.

2.2 Finding Potential Bits of A Word

This is the first-level grouping to identify potential bits which may belong to the same word. We first look at the netlist file. Each line describes a net which is usually fanout of a gate. Each net is compared against the next line in the netlist file. For two nets in adjacent lines, if the roots of their fanin-cones is the same gate type they will be grouped together as potential bits. For example, in the example of Figure 1, assume that we initially had five nets (`U215`, `U216`, ...`U219`) in consecutive lines. They all had 3-input NAND gates in the roots of their fanin-cones. (This can be seen in the figure for bits `U215`, `U216`, `U217`.) So these five nets were grouped together as potential bits at this stage.

Using the above grouping, we observed that often times, a group of bits may be larger in size than the actual word sizes in the circuit; for example a group at this stage may contain multiple words, or it may contain one word with some individual bits not belonging to any word in that group.

The above procedure may result in a word to be broken among two groups. However, the main benefit of this procedure is that it is extremely fast; it requires visiting each line in the netlist file once and doing a quick comparison to check the gate type at the root with its adjacent line. We leave developing efficient procedures for cross-checking among adjacent groups to a future improvement of this work. We note prior work [6] also used a similar distance-based strategy within a netlist file for grouping of potential bits. We note other variations of distance-based strategies which are not dependent on the netlist can also be alternatively used and like to emphasize that the goal of this stage is only to provide a rough grouping of potential bits.

2.3 Finding Partially-Matching Structures

Each group created from the previous step is visited and the potential bits are further grouped into smaller subgroups that are more likely to belong to the same word. For one group, the output of this stage is a set of subgroups where a subgroup may just be a single bit. For each subgroup, we also remember a list of dissimilar subtrees among its bits where each subtree is remembered by the name of the net at its root. In our example, assume the initial group of `U215` to `U219` is divided into a subgroup of `U215` to `U217`, and another subgroup of `U218` and `U219`. For the subgroup of `U215` to `U217`, there are three dissimilar subtrees which are shown by the red dashed circle in Figure 1. Each dissimilar subtree is only recorded by the name of its root.

Next we explain how the above-described subgroups are formed at this stage. We start with the first bit in the group and create a subgroup for it. Next, we visit the second bit and compare it with the first bit. If these two bits have a fully or partially -matching structure, they will be grouped into the subgroup of the first bit. Otherwise a new subgroup is created for the second bit. The process continues by visiting the remaining bits sequentially, comparing each bit with its previous one.

Note, we don't check if a bit may be fully or partially matching with the rest of the subgroups. For example, if bit `U216` is in the same subgroup as `U217` because of a partial match, it may also partially match (but in a different subtree) with the subgroup of `U218`. But to ensure each bit belongs only to one subgroup, we group `U217` in only one subgroup, and for efficiency, we set this to be the group of `U216` which is the bit adjacent to it.

The core step in the above procedure is efficiently determining if two bits are partially matching in their fanin-cones which is done as follows. We store each bit as a union of its second-level subtrees. For example in Figure 1, for each bit such as U215, there are three subtrees at the second-level. We use an efficient procedure which effectively checks all combinations of subtrees among different bits to declare if they are similar to dissimilar. If two bits are similar in all their subtrees, then they have fully matching fanin-cones. Otherwise if they have at least one matching subtree, they have partially-matching fanin-cones.

Next, we discuss for two bits, how we check similarity for all subtree combinations among them. First, each subtree of each bit is recorded as a string obtained by doing a post-order traversal from its root to its leaves. Note, each entry in the post-order traversal only records the gate type of the corresponding node in the subtree. Multiple fanins of a gate are sorted lexicographically.

The string representation of a tree using post-order traversal was done in early prior work. (For example it was referred to as Polish Expression in chip floorplanning [12].) A similar representation is used in [6] and referred to as a hash key. In this work, we also use the term hash key to refer to the post-order traversal of a subtree. To check if two trees have a similar structure, we check if their hash keys are equal which does not consider all possibilities but is much faster than the accurate procedure of graph isomorphism.

For each bit, we further sort the hash keys of its subtrees. A pointer is created which initially points to the beginning of this hash key list for each bit. To compare subtree combinations of different bits, we start by comparing the hash keys pointed by current pointer locations. For example, the hash key pointed in the first bit is compared with the hash key pointed in the second bit. If they are equal, the two subtrees are structurally similar. So the pointer in the second hash key is incremented and the procedure is repeated. If they are not equal, it is not necessary to check further down the list and only the first pointer gets incremented. This allows not to cross-check all subtree combinations between two bits because the hash keys are traversed in sorted order. The process terminates when the two pointers reach the end of the hash key lists and each hash key is only visited once.

2.4 Finding Relevant Control Signals

The previous step created subgroups of potential bits which are ensured to have a fully or partially matching fanin-cone structures. Each bit is stored as the union of its hash keys (representing its second-level subtrees). In this step, we search for relevant control signals for each dissimilar subtree of each bit in a subgroup. Recall, the dissimilar subtree(s) of a bit are already identified in the previous step.

To identify what we consider as relevant control signals, we first list the set of nets which are common among all the dissimilar subtrees. Next, from this set, we remove the ones which are in the fanin-cones of the other nets in the set. For example, in Figure 1, for the three dissimilar subtrees, the red circle identifies the common portion. Net U223 is common to each dissimilar subtree. However, it is in the fanin cone of U201 which is also in the common region so U223 is removed. This is because the impact of U223 for circuit reduction in the subsequent step is fully captured by U201. So the relevant control signals will be U201 and U221 in this case.

Note, we do not consider control signals that only appear in matching subtrees such as as U202*, as they cannot help create additional structural similarity and would only increase complexity. In contrast, the control signals identified by our technique are in the dissimilar subtrees. They are likely to result in circuit reduction into similar subtrees if assigned to proper values which helps identify words with partially matching structures.*

2.5 Circuit Reduction and Word Identification

Once control signals for a potential word have been identified, we try assigning them to constant values and simplify the circuit based on the assignment(s). This is done iteratively. First, we consider assigning a value to only one control signal at a time. The assigned value to a control signal will be the controlling value to one of the logic gates that the control signal is feeding into. For each assignment, we simplify the circuit as much as possible. This is done by propagating the values forward and backwards throughout the netlist. After all net assignments have been inferred, assigned nets and gates with assigned outputs are removed. If a gate has only a single input remaining, it is reduced appropriately into either a buffer or inverter. After the netlist is simplified we recheck if words can be identified by now looking for fully-similar structures (by checking the hash keys of all subtrees among the bits). If no word can be identified, we then consider feasible assignments to any two identified control signals within a potential word. In practice the number of control signals in a potential word is small so exploring all combinations is also done very quickly.

For example in Figure 1, we first assign control signal U221 to 0 which is the controlling value to the 2-input NAND gates that it feeds into. This results in removing the dashed subtrees of bits U215 and U216 but not the dissimilar subtree in U217. Next, we examine the case when control signal U201 is set to controlling value of 0. This results in circuit simplification which eliminates all the dissimilar subtrees in the 3 bits, effectively turning them into fully-similar structures after the simplification which will then identify the potential word as an actual 3-bit word. Checking for similarity is done by checking the equality of the sorted hash key lists stored for each bit after circuit reduction. The procedure is similar to the one listed in Section 2.3.

2.6 Runtime Complexity

The first step to find potential bits of a word (Section 2.2) takes $O(N)$for a netlist with N nets. To find partially-matching structures, for two bits i and j with k_i and k_j subtrees respectively, we require $O(k_i + k_j)$ to compare their hash key combinations in order find the dissimilar subtrees. Generating the hash key of each subtree requires a post-order traversal which has a runtime complexity linear in the number of nodes and edges of the subtree. Finding control signals requires checking for common nets among the already-identified dissimilar subtrees. It needs traversing the dissimilar subtrees once. We observed the number of relevant control signals involved in a word are limited so considering circuit simplification based on assignment to one control signal, and then all pairs of control signals is also fast in practice. Finally identifying the words for a simplified circuit also requires checking the hash keys in the simplified subtrees as was explained before. In our experiments including a circuit with more than 100K gates, we report runtime of at most a few minutes.

Table 1: Comparison of our approach with the shape hashing word identification technique [6].

	Benchmark Info			Reference							
	#gates	#nets	#FFs	#Words	#Bits	Technique	Full Found (%Word)	Partial Found (Word Frag. Rate)	Not Found (%Words)	Time(s)	#Control Signals
b03	122	156	30	7	3.14	Base	71.4	0.67	14.3	0.00	0
						Ours	85.7	0.00	14.3	0.01	1
b04	652	729	66	9	7.33	Base	77.8	0.50	11.1	0.01	0
						Ours	88.9	0.00	11.1	0.02	1
b05	927	962	34	5	6.20	Base	80.0	0.00	20.0	0.00	0
						Ours	80.0	0.00	20.0	0.03	0
b07	383	433	49	7	7.00	Base	57.1	0.33	14.3	0.00	0
						Ours	57.1	0.33	14.3	0.00	1
b08	149	179	21	5	4.20	Base	40.0	0.58	20.0	0.00	0
						Ours	80.0	0.00	20.0	0.01	3
b11	726	764	31	5	6.20	Base	60.0	0.54	0.0	0.00	0
						Ours	60.0	0.54	0.0	0.01	0
b12	944	1070	121	46	2.52	Base	82.6	0.50	8.7	0.01	0
						Ours	91.3	0.30	4.3	0.09	7
b13	289	352	53	7	5.29	Base	28.6	0.75	28.6	0.00	0
						Ours	42.9	0.60	14.3	0.02	2
b14	9767	10044	245	8	30.13	Base	50.0	0.13	0.0	0.01	0
						Ours	62.5	0.08	0.0	0.65	4
b15	8367	8852	449	32	13.69	Base	68.8	0.19	6.3	0.01	0
						Ours	81.3	0.24	0.0	0.31	4
b17	30777	32229	1415	98	14.06	Base	69.4	0.18	6.1	0.05	0
						Ours	74.5	0.23	1.0	20.53	18
b18	111241	114589	3320	212	15.28	Base	52.8	0.20	5.7	0.15	0
						Ours	58.5	0.22	4.7	215.99	36
Average						**Base**	**61.54%**	**0.381**	**11.25%**	**0.021**	
						Ours	**71.89%**	**0.213**	**8.67%**	**19.806**	

3. SIMULATION RESULTS

We implemented the procedure given in Section 2 in C++. All experiments ran on a Ubuntu 14.04 Linux machine with an Intel i7-4910MQ and 16 GB of memory. We experimented with the ITC99 benchmark circuits [1]. Both gate-level and RT-level implementations of these benchmarks were downloaded[2].

We used the following setting to create a golden reference case with already-verified words which allowed us to experiment with many ITC99 benchmarks. We utilized the observation that register names in the VHDL code for each benchmark were preserved in the gate-level netlist file. Specifically, the output net of each flip-flop is named using the register name and bit position it corresponds to. Using this information, we first manually group all bits of each register with matching names across VHDL and gate-level netlist into reference words. In our experiments, these words are the input nets to the flip-flops, rather than the named output nets, since we are matching structure based on fanin-cones.

In Table 1, columns 5 and 6, we list the number of reference words that we identified using the above procedure as well as the average size of the reference words for each benchmark. As can seen in column 4, the number of flip-flops in each benchmarks in many cases is close to the number of words multiplied by the average word size in that benchmarks. We only experimented with those ITC benchmarks with at least 5 identified reference words.

We compare our strategy against the shape-hashing algorithm given in [6], which is referred to in Table 1 as Base. Since we did not have access to the source code, we wrote our own implementation. Shape-hashing uses similar techniques to our approach, but only considers the un-simplified structure of the netlist when grouping bits into words. It also only groups bits which have a fully-matched structure into bits. Given that the novelty of our approach is in translating the structure of the netlist for matching and also match based on partially-matched structures, comparison with shape-hashing identifies the additional accuracy gained by our technique. For each benchmark in Table 1, we evaluate both shape-hashing and our technique using the following three metrics based on the number of words that were fully found, partially found, and not found.

First and most importantly, we measure the number of words that are fully found. We consider a reference word to be fully found if a word found using our technique includes all bits of the reference word. In Table 1 column 8, we report the number of words that are fully found as a percentage with respect to the number of words in the reference case for both techniques. On average our technique identified 71.89% of the reference words. Our approach found on average 10% additional words compared to the base case. Over all benchmarks we observe that our technique never performs worse than the base case and it always finds at least as many full words. This amount of improvement can be considered quite valuable because it is also used in the subsequent stages of reverse engineering techniques such as word propagation in [6] which require an initial set of full words to operate on. Having a larger set of full words will allow these functions to achieve better results.

For our second metric, we measure the number of words that were not found. A reference word is considered not found if none of the bits from any reference word are grouped together as a word generated by a technique. In other words, each bit of a reference word appears in a different word in the generated word set of our technique. This is an important metric, as without any grouping of bits, no additional information has been gained about this word. As can be seen from Table 1, our technique has a lower percentage of words that are not found compared to the base case (11.25% versus 8.67%). Also, our technique never performs worse than the base case in this metric as well.

[2]http://www.cad.polito.it/downloads/tools/itc99.html

In general, words that are not found are state or other types of control registers. The individual bits in these registers often show little or no structural similarity since individual bits may serve different purposes in the circuit. In some cases our technique manages to utilize the little structural symmetry that can exist in control words, but rarely to fully complete the word, providing small improvement over the base case.

Last we investigate words that are partially found. These are those reference words that do not fall into the fully-found and not-found categories. For a given identification technique, we define a reference word to be partially found if at least some but not all of the word's bits are grouped in the generated words using that technique.

For comparison purposes, we calculate a fragmentation rate for each partially-found word. This is the number of generated words that a reference word's bits are spread across. For instance, an 8-bit reference word split into two 4-bit generated words would be fragmented into two pieces. In our results, we normalize fragmentation of a reference word by the number of bits in the word, and average this normalized fragmentation across all partially-found words. Using this metric, a smaller value implies smaller average fragmentation in partially-found words. Also an average fragmentation of 0 indicates there were no partially-found words which can be seen is the case for our technique in benchmark `b04` while the base case has an average fragmentation rate of 0.5 in its partially-found words. As shown in Table 1, we find our technique provides consistent improvement to this metric compared to the base case over all benchmarks.

We note in several benchmarks such as `b04`, the percentage of reference words that were not found is the same in our approach and the base case. However in the same benchmarks we often report a lower fragmentation rate or a higher percentage of fully-found words using our technique.

Next, in Table 1, we also include the runtime for each technique in seconds. This is not done for comparison as the two techniques do very different levels of analysis. What it does show is that the runtime complexity for our technique remains reasonable even for large benchmarks. For b18, our largest benchmark with over 100K nets and gates, our technique finished in a little over three and a half minutes, which is reasonable for analysis on a circuit of this size.

Finally, in the last column of Table 1, we include the number of relevant control signals found for each benchmark using our approach to illustrate a few scenarios of our technique. In benchmark `b15`, four control signals were found by our approach which resulted in finding four additional full words compared to the base case (22 versus 26 full words). In this case, our techniques works exactly as desired. Each control signal found was useful and capable of uncovering one complete word which otherwise would have been left undetected in the base approach.

Next, analyzing benchmark `b18`, our technique found 36 relevant control signals for a gain of twelve additional full words. Recall that after finding control signals, for efficiency, we assign values to up to two of them simultaneously. While some words in this benchmark utilized assignment to two control signals, there were cases of potential words which may have been improved if more than two control signals were simultaneously assigned to simplify the netlist. This is an improvement that we plan to realize as part of the future extensions of this work.

4. CONCLUSIONS

We presented a procedure for word-level identification in a gate-level netlist which can be beneficial to the most challenging variation of reverse engineering problem when there are no information about high-level design specifications and potentially-used design templates. The main novelty of our procedure is matching bits belonging to the same word based on partially-matching structures which was done by discovering and utilizing relevant control signals.

In our experiments we created a reference case that accurately identified a subset of actual words for each benchmark which enabled us to validate our techniques using many benchmarks. We reported consistent improvements in three complementary metrics to measure the quality of our technique based on words that were fully found, partially found, and not found. These improvements are valuable because word identification can serve as the very first stage of gate-level reverse engineering techniques so having a higher number of identified words allows the existing technique to achieve better results.

5. REFERENCES

[1] F. Corno, M. S. Reorda, and G. Squillero. Rt-level itc 99 benchmarks and first atpg results. *IEEE Design & Test of Computers*, 17(3):44–53, 2000.

[2] M. C. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers*, 16(3):72–80, 1999.

[3] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *IEEE Symposium on Security and Privacy*, pages 159–172, 2010.

[4] R. K. Jain, P. Tiwari, and S. Ghosh. Automated determination of top level control signals. In *Design, Automation and Test in Europe*, pages 509–512, 2013.

[5] Y. Jin, N. Kupp, and Y. Makris. Experiences in hardware trojan design and implementation. In *International Workshop on Hardware-Oriented Security and Trust*, pages 50–57, 2009.

[6] W. Li, A. Gascón, 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 *International Symposium on Hardware-Oriented Security*, pages 67–74, 2013.

[7] W. Li, Z. Wasson, and S. A. Seshia. Reverse engineering circuits using behavioral pattern mining. In *International Symposium on Hardware-Oriented Security and Trust*, pages 83–88, 2012.

[8] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri. Security analysis of logic obfuscation. In *Design Automation Conference*, pages 83–89, 2012.

[9] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascón, W. Y. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik. Reverse engineering digital circuits using structural and functional analyses. *IEEE Transactions on Emerging Topics in Computing*, 2(1):63–80, 2014.

[10] M. Tehranipoor and F. Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design & Test of Computers*, 27(1):10–25, 2010.

[11] S. Wei, S. Meguerdichian, and M. Potkonjak. Gate-level characterization: foundations and hardware security applications. In *Design Automation Conference*, pages 222–227, 2010.

[12] D. F. Wong and C. L. Liu. A new algorithm for floorplan design. In *Design Automation Conference*, pages 101–107, 1986.

[13] J. Zhang and Q. Xu. On hardware trojan design and implementation at register-transfer level. In *International Symposium on Hardware-Oriented Security and Trust*, pages 107–112, 2013.

[14] X. Zhang and M. Tehranipoor. Case study: Detecting hardware trojans in third-party digital IP cores. In *International Symposium on Hardware-Oriented Security and Trust*, pages 67–70, 2011.