Reverse Engineering Digital Circuits Using Functional Analysis

Pramod Subramanyan[†], Nestan Tsiskaridze, Kanika Pasricha, Dillon Reisman, Adriana Susnea and Sharad Malik
Departments of Electrical Engineering and Computer Science, Princeton University.

Abstract—**Integrated circuits (ICs) are now designed and fabricated in a globalized multi-vendor environment making them vulnerable to malicious design changes, the insertion of hardware trojans/malware and intellectual property (IP) theft. Algorithmic reverse engineering of digital circuits can mitigate these concerns by enabling analysts to detect malicious hardware, verify the integrity of ICs and detect IP violations.**

In this paper, we present a set of algorithms for the reverse engineering of digital circuits starting from an unstructured netlist and resulting in a high-level netlist with components such as register files, counters, adders and subtracters. Our techniques require no manual intervention and experiments show that they determine the functionality of more than 51% and up to 93% of the gates in each of the practical test circuits that we examine.

I. Introduction

Contemporary integrated circuits (ICs) are designed and fabricated in a globalized, multi-vendor environment due to which ICs are vulnerable to malicious design changes and the insertion of hardware trojans and malware. The possibility that malicious chips might be used in sensitive locations such as military, financial and government infrastructure is a serious and pressing concern to both the users and designers of contemporary ICs [6, 10, 14]. For example, the DARPA IRIS program seeks to develop techniques for reverse engineering digital, analog and mixed-signal ICs to determine their integrity for use in sensitive installations [4]. Algorithmic approaches to reverse engineering chips can aid in the detection of hardware trojans, malicious design changes and in verifying the integrity of untrusted design components for which trustworthy source code may not be available. Reverse engineering is important in detecting intellectual property violations, also considered a "serious concern" for the semiconductor industry [12].

In this paper, we study a portfolio of fully algorithmic approaches to reverse engineer digital circuits. We start our analysis with an unstructured netlist with the objective of inferring a high-level netlist with components such as register files, adders and counters. The key challenge in analyzing an unstructured netlist is that *we have no information about the boundaries of the modules* that comprise the netlist. Therefore, we tackle the reverse engineering problem through a variety of algorithms that "carve out" portions of the netlist to *generate potential/candidate modules* and employ techniques similar to those used in design synthesis and verification to determine the functionality of these modules.

A. Related Work

Fully algorithmic reverse engineering is not a well-studied problem. Previous work primarily suggests strategies of attack for a human analyst [7, 15]. For example, in their investigation of the ISCAS '85 benchmarks, Hansen et al. analyze replicated structurally isomorphic blocks [7]. The cut-based Boolean matching and aggregation algorithms presented in §II-A and §II-B are generalizations of this idea.

A recent attempt at addressing the reverse engineering problem algorithmically is by Li et al. [9]. They present a method for behavioral matching of an unknown sub-circuit against a library of *abstract components* but assume that methods are available to generate sub-circuits from the unstructured netlist. Therefore, our set of solutions is complementary to theirs because: (a) we target different kinds of components for reverse engineering and (b) we analyze an unstructured netlist as opposed to sub-circuit matching.

An alternative approach to malware detection relies on comparing side channel signals such as power [1], current [16] and timing [8] between the trusted and untrusted versions of the design. The assumption is that a trusted version of the design is available. This may not be true for untrusted component IPs. Furthermore, these do not target IP violations.

B. Solution Overview

The objective of our work is deriving a useful high-level description of the circuit from an unstructured netlist. In particular, we focus on reverse engineering *datapath elements* in digital circuits because the majority of the gates in microprocessor-like designs are part of the datapath. Datapath elements are also suitable for algorithmic analysis due to their regularity and high-degree of replication. Even when focusing primarily on the datapath, reverse engineering is still a very hard problem because we are starting with a sea of gates and it is not obvious how to go about finding some meaningful subset of the gates/latches for algorithmic analysis. Hence, our approach integrates a number of different techniques which tackle different aspects of the problem. Figure 1 shows the techniques we introduce and their inter-relationships.

Our strategy is to attack the problem in two stages. The first stage identifies *potential module boundaries* using topological/functional analyses. The second stage *functionally analyzes potential modules* to understand their behavior.

The contributions of this paper are as follows:

1) We present a novel application of cut-based Boolean matching to find replicated bitslices[1] in the netlist. This analysis helps us find circuit nodes that correspond to functions such as 1-bit adders and 1-bit multiplexers.
2) We present algorithms that topologically analyze the results of bitslice matching to *aggregate* multibit components such as multiplexers, adders and subtracters.

[1]In this paper, a bitslice is a Boolean function with one output and a small number of inputs that is replicated to construct multibit datapath operators.

[†]*Corresponding author e-mail: psubrama@princeton.edu. This research was supported through a grant from DARPA as part of the IRIS project.*
978-3-9815370-0-0/DATE13/©2013 EDAA

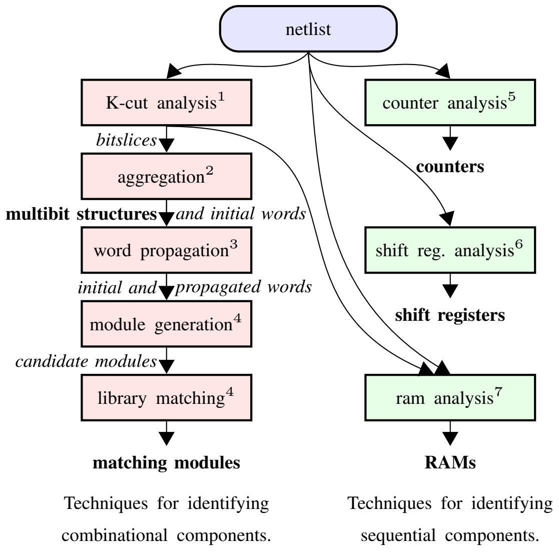

Fig. 1. Portfolio of the reverse engineering techniques introduced in this paper. The figure shows the dependences of the techniques and their outputs. Outputs in **bold** are **results** of our inference tool. Outputs in *italics* are *hints* which may used as a starting point for investigation by a human analyst. Superscripts refer to items in our list of contributions.

3) Analyzing aggregated modules helps identify *bits which are operated upon simultaneously*, allowing us to infer *words*. These inferred words are then used in our *word propagation algorithm* to generate additional words.
4) Our *module generation algorithm* analyzes words which are structurally connected to generate *candidate unknown modules*. These are potential operators with word arguments and results and are matched against a component library using permutation and phase independent Boolean matching [11].

5,6) We present *algorithms to identify counters and shift registers* using topological analyses combined with a satisfiability (SAT) checking formulation.

7) We present novel *algorithms that identify register files and RAM arrays* BDD-based functional analysis.

We evaluate our algorithms by experimenting with eight unstructured netlists.[2] These netlists were obtained by synthesizing designs from opencores.org and other sources. The largest of these (RISC FPU) has 14291 gates and 3097 latches/flip-flops. Results show that our inference algorithms can completely determine the functionality of more than 51% and up to 93% of gates in a fully automated manner.

II. Bitslice Identification and Aggregation

Our first algorithm is based on the observation that many datapath elements consist of replicated bitslices connected in a specific topology.

A. Bitslice Identification

The goal of bitslice identification is to identify all nodes in the circuit that match elements in a bitslice library. For instance, we might be interested in finding all nodes that match the adder carry function $ab + bc + ca$, this might help identify multibit adders. We adopt a *functional* matching approach, which matches based on the function implemented by a set of gates instead of matching structural patterns. This uses cut-enumeration and Boolean matching, which was initially introduced for technology mapping [3, 2].

A *feasible cut* of a circuit node G is defined as a set of nodes in the transitive fan-in cone of G such that a consistent assignment of truth values to each node in the set completely determines the value of G [2]. A cut is said to be *k-feasible* if it has no more than k inputs. The trivial cut $\{G\}$ is always *k-feasible*. The set of *k-feasible* cuts for a gate is recursively computed by enumerating the union of all *k-feasible* cuts of the gate's inputs such that this union has k or fewer inputs.

Our tool enumerates all 6-feasible cuts. We found that the average number of 6-feasible cuts per gate is between 15 and 35. The number of cuts for $k > 6$ is significantly higher.[3] Although we are restricted to bitslices with six or fewer inputs, this is not a major limitation as most common bitslices have less than six inputs; e.g., a full adder bitslice has 3 inputs.

Once all cuts are identified, they are grouped into equivalence classes using permutation-independent Boolean matching. For example, nodes matching the function $y = ab + c$ and nodes matching $y = bc + a$ are grouped into the same class. Each equivalence class may match a known library function.

B. Aggregation to Multibit Components

Now that we have all the nodes that match a particular function, the next step is to look for matching nodes connected in interesting patterns. *Aggregating* replicated bitslices which are connected in specific patterns is our first technique for identifying combinational modules. The following subsections expand on our aggregation algorithms.

1) Common Signals in Replicated Bitslice: This algorithm considers all bitslices that match a particular function and groups them using common input signals. For instance, consider the function that represents a 2:1 multiplexer: $f(a, b, s) = sa + \neg sb$. Here we group all matching bitslices which have a common select signal (s in this example). Common signal aggregation finds 185 decoders and 210 multiplexers in our largest test article (*RISC FPU*).

2) Propagated Signal(s) in Replicated Bitslices: In this case, the algorithm considers all bitslices matching a particular function such that the output of one bitslice is the input of another (e.g., carry chain in a ripple carry, parity tree). Propagated signal aggregation finds 129 adders/subtracters and 102 parity trees in the *RISC FPU* test article.

C. Word Identification and Word Propagation

Aggregated bitslices tell us about circuit nodes that are operated upon simultaneously. These nodes are likely to form part of same *word*. Our tool groups the bits that are inputs/outputs of aggregated modules into "word" data structures.

Once a few words are identified, more words can be generated by *propagating* known words across structurally

[2]The netlists we experiment with are: RISC FPU, oc8051, mips16, ae18, cpu8080, Open8, eVoter and router.

[3]These observations are in line with published work on cut enumeration [2].

identical gates. This analysis examines the inputs and outputs of gates that form words for structural symmetry and then propagates words from inputs to outputs and from one set of input terminals to others. The *RISC FPU* test article had 758 initial words and 2461 propagated words.

D. Module Identification and Matching

The two main limitations of bitslice identification are: (i) we are limited to bitslices with a maximum of 6 inputs due to the $k \leq 6$ limitation on cut-enumeration and (ii) it is difficult to identify combinational structures that do not have a clean interconnection pattern. Our second approach overcomes these limitations by constructing entire modules and then matching them against a component library.

The intuition here is that since datapath circuits operate on word inputs and produce word outputs, cutting out portions of the circuit that exist *between words* may find interesting candidate modules. Our module identification algorithm operates in three steps. First, a *word connection graph* is created that formalizes the structural connectivity between words. Then, *candidate unknown modules* are created using the gates in between words. These modules are then compared with a library of known modules using BDD-based permutation and phase independent Boolean matching [11].

III. Identifying Sequential Components

A reverse engineering solution must identify commonly occurring sequential components such as RAM arrays, register files, counters and shift registers because these cover a significant number of gates in circuits and also give insight into functionality of the circuit. The challenge here is again in finding meaningful module boundaries for these components given the unstructured netlist. Our strategy is to devise topological analyses to find circuit nodes that are *potential* counters, RAM outputs or shift registers. We then formulate functional analyses using SAT and BDDs that verify correctness of the "guess" made by the topological analysis. The rest of this section presents algorithms to identify RAM arrays/register files, counters and shift registers.

A. Counter Identification

The specific problem here is to identify sets of latches in the unstructured netlist that are counters. We first observe that the topology of signal flow between the latches in a counter is as shown in Figure 2.

Based on this observation, our analysis is performed in two stages. First, potential counters are generated by finding sets of latches whose interconnections match the counter topology. The next step uses a SAT-based functional analysis to verify whether the functions at the inputs of the latches in the counter satisfy the following conditions: (i) each latch toggles either when all the low-order latches are 1 (up counter) or all the low-order latches are 0 (down counter) and (ii) the conditions that control when the counter is enabled/reset are the same for all the bits of the counter. Five counters were found in the *oc8051* test article.

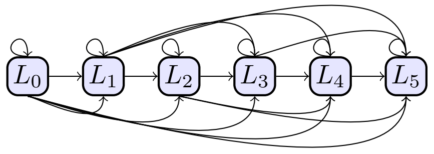

Fig. 2. Latch-to-latch information flow in a counter: each latch in the counter is driven by the latches corresponding to the lower-order bits.

B. Shift Register Identification

As with counters, our goal here is to identify the set of latches and associated logic that form shift registers given an unstructured netlist. The shift register identification algorithm is similar to the counter identification algorithm in that it uses a topological check and a SAT formulation. Only the topology and verification conditions differ.

Shift registers may consist of multiple bits shifting in tandem from one set of latches to another. The basic algorithm finds each cascading chain of latches as separate shift registers. To *aggregate* shift registers, first we group shift registers by length. Next, we form equivalence classes within each group where shift registers with the same set, reset and shift-enable functions are classified together. Finally, each equivalence class is output as a multibit shift register module. Seven shift registers were found in the *RISC FPU* test article.

C. Identifying RAMs

This section targets small RAM arrays and register files. Our objective here is to find the latches/flip-flops that form the RAM, associated logic that reads data (called "read-logic") and logic that writes data into the latches (called "write-logic").

The first step in detecting the "read-logic" is to identify logic trees that whose leaves are latches. After this a BDD-based analysis checks whether every combination of address signals propagates the output of exactly one of the latches to the root of the tree. Logic trees with identical address signals are aggregated to form multibit RAM modules.

To identify the "write-logic", the cut-based analysis is used to find the write-enable signal for each latch in the RAM. Next, the common support[4] of all the write-enable signals is computed. A BDD-based analysis is then used to verify properties that correspond to proving that (i) every latch can be written to and (ii) no two latches which are not part of the same word can be written to simultaneously. One dual-ported register file was found in the *RISC FPU* using this algorithm.

IV. Experimental Results

A. Methodology

We developed an inference tool using the C++ programming language that implements the algorithms described in Sections II and III. The tool takes as input a synthesized verilog netlist, analyzes it and outputs an abstracted netlist with the inferred

[4]We first compute the full-combinational fanin-cone for every write-enable signal. The intersection of each these cones (called the common cone) is then computed. For each write-enable signal, the set difference between the full-combinational fanin-cone and the common cone yields the common support.

TABLE I
COVERAGE RESULTS

Design Information					**Bitslice identification and aggregation**							**Sequential components**			**Coverage and execution time**		
Design	i/p	o/p	gate	latch	a/s	dec	dm	eq	gf	mux	lt	ram	sr	cnt	cov	time	notes
router	35	26	944	182	0	28	3	0	10	46	2	0	0	4	53 %	4s	on-chip router
eVoter	31	15	1291	109	0	23	1	10	7	4	26	0	0	0	51%	10s	vot. mac. controller
Open8[†]	19	26	1807	237	22	30	1	1	21	42	20	1	1	2	66 %	33s	Open8 μRISC
cpu8080[†]	12	29	2258	243	7	55	7	0	30	77	21	0	0	0	59 %	60s	8080 CPU
ae18[†]	32	64	3466	1094	25	50	3	0	28	113	32	0	3	10	70 %	25s	PIC μcontroller
mips16[†]	1	8	6986	4380	2	93	0	0	9	289	51	1	1	27	93 %	31s	16-bit MIPS CPU
oc8051[†]	86	78	8093	2748	8	539	15	6	60	407	201	4	3	5	70 %	242s	8051 μcontroller
RISC FPU	35	66	14291	3097	129	171	3	31	97	189	194	1	7	3	85 %	529s	32-bit RISC FPU
Total			39136												78%	934s	[†]from opencores.org

Legend for table header: i/p: chip inputs; o/p: chip outputs; gate: number of gates; latch: number of latches a/s: adders/subtracters; dec: decoders; dm: demultiplexers; eq: equality comparators; gf: gating functions (and2/or2 etc. of a word with a common signal); mux:multiplexors; lt: parity tree, zero-detect and one-detect; ram:RAMs/register files; sr: shift registers; cnt: counters; cov: coverage in percentage of gates covered; time: execution time.

components. The tool uses the CU Decision Diagram (CUDD) Package version 2.4.2 for the BDD-based analyses [13]. MiniSat version 2.2 was used for satisfiability checking [5].

Experiments were performed on an Intel® Xeon® E31230 CPU clocked at 3.20GHz with 32 GB of RAM. We used eight netlists for our experiments. Source code for five of these was obtained from opencores.org and these are marked with a dagger in Table I. All the designs were synthesized using an IBM/ARM cell library for a 45nm SOI process.

B. Summary of Results

Table I shows the modules identified and overall coverage obtained using our inference algorithms. Coverage is measured as a percentage of gates in the design which are covered by inferred modules. The table also shows information about the netlists being analyzed, the number of inferred modules of various types and the execution time of the tool.

For the three biggest netlists, coverage is above 75% and reaches up to 93% for the 16-bit MIPS CPU. These netlists all have a large number of replicated bitslices in the datapath which are captured well by the bitslice identification and aggregation algorithms. In contrast, the smaller netlists have a significant fraction of gates devoted to irregular control logic, which is hard to identify in a fully automated solution. When considering coverage across all the designs, about 78% of the 39136 gates analyzed were identified by our algorithms and the total execution time was 15 minutes and 34 seconds.

Sometimes a gate might be placed in multiple inferred modules. In the *oc8051* design, the RAM read-logic consists of a number of muxes which are identified by the bitslice aggregation algorithms and the RAM analysis algorithm. To resolve this, we define a *dominance* relation between inferred modules. One inferred module *dominates* another if the latter is fully contained inside the former. Dominated modules are eliminated from the output of the tool. Even after the elimination of dominated modules, some "conflicts" remain. We expect that a human analyst will resolve these.

V. CONCLUSION

Integrated circuits are now designed and fabricated in a globalized and multi-vendor environment making them vulnerable to malicious design changes and hardware trojans. Algorithmic reverse engineering can mitigate these risks by helping detect malware and verify the integrity of critical ICs.

The key challenge in reverse engineering digital circuits is generating meaningful module boundaries given a very large unstructured netlist of gates. In this paper, our main contribution is a portfolio of algorithms for reverse engineering that: (i) find module boundaries for a variety of combinational and sequential components and (ii) functional analyses that verify the behavior of these modules. Experiments showed that the functionality of 51% to 93% of the gates in a netlist may be automatically inferred using our algorithms.

REFERENCES

[1] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan detection using ic fingerprinting. In *Proc. of IEEE SP 2007.*

[2] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam. Reducing Structural Bias in Technology Mapping. In *ICCAD 2005.*

[3] J. Cong and Y. Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE Trans. on Comp.-Aided Des. of Integrated Circuits and Sys.*, 1994.

[4] DARPA. Integrity and Reliability of Integrated Circuits (IRIS). http://www.darpa.mil/Our_Work/MTO/Programs/Integrity_and_Reliability_of_Integrated_Circuits_(IRIS).aspx, 2012.

[5] N. Eén and N. Sörensson. An Extensible SAT-solver. In *SAT 2003.*

[6] Defence Science Board Task Force. High Performance Microchip Supply. http://www.acq.osd.mil/dsb/reports/ADA435563.pdf, 2005.

[7] M.C. Hansen, H. Yalcin, and J.P. Hayes. Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering. *IEEE Design & Test of Comp.*, 16(3):72 –80, 1999.

[8] Y. Jin and Y. Makris. Hardware Trojan Detection Using Path Delay Fingerprint. In *Proc. of HOST 2008.*

[9] W. Li, Z. Wasson, and S. A. Seshia. Reverse Engineering Circuits Using Behavioral Pattern Mining. In *Proc. of HOST 2012.*

[10] J. Lieberman. National Security Aspects of the Global Migration of the U.S. Semiconductor Industry. White paper, Airland Subcommitte, US Senate Armed Services Committee, 2003.

[11] J. Mohnke and S. Malik. Permutation and Phase Independent Boolean Comparison. *Integration, the VLSI Journal*, 16(2), December 1993.

[12] SEMI. IP Challenges for the Semiconductor Equipment and Materials Industry. http://www.semi.org/sites/semi.org/files/docs/2012_IP_White_Paper.pdf, 2012.

[13] F. Somenzi. CUDD: CU Decision Diagram Package. http://vlsi.colorado.edu/~fabio/CUDD/, 2011.

[14] M. Tehranipoor and F. Koushanfar. A Survey of Hardware Trojan Taxonomy and Detection. *IEEE Design & Test of Comp.*, 2009.

[15] R. Torrance and D. James. The State-of-the-Art in IC Reverse Engineering. In *Proc. of CHES 2009*, 2009.

[16] X. Wang, H. Salmani, M. Tehranipoor, and J. Plusquellic. Hardware Trojan Detection and Isolation Using Current Integration and Localized Current Analysis. In *Proc. of DFT 2008*, 2008.