Reverse Engineering Digital Circuits Using Structural and Functional Analyses

Pramod Subramanyan, Nestan Tsiskaridze, Wenchao Li, Adrià Gascón, Wei Yang Tan, Ashish Tiwari, Natarajan Shankar, Sanjit A. Seshia and Sharad Malik

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 subtractors. Our techniques require no manual intervention and experiments show that they determine the functionality of more than 45% and up to 93% of the gates in each of the test circuits that we examine. We also demonstrate that our algorithms are scalable to real designs by experimenting with a very large, highly-optimized system-on-chip (SoC) design with over 375,000 combinational elements. Our inference algorithms cover 68% of the gates in this SoC. We also demonstrate that our algorithms are effective in aiding a human analyst detect hardware trojans in an unstructured netlist.

1 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 [8, 15, 28, 1, 17]. 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 [5]. Previous work primarily suggests strategies of reverse engineering and in verifying the integrity of untrusted design components for which trustworthy source code may not be available. Reverse engineering is also important in detecting intellectual property violations, considered a “serious concern” for the semiconductor industry [7].

In this paper, we study a portfolio of fully algorithmic approaches to reverse engineer digital circuits. We analyze 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 contained in 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. In particular, this paper focuses on algorithmic reverse engineering of datapath components in an unstructured netlist. The objective is to aid a human analyst understand the functionality of an unstructured netlist by algorithmically identifying as many components as possible.¹

1.1 Related Work

Fully algorithmic reverse engineering is a relatively new field of research. Previous work primarily suggests strategies of attack for a human analyst [9, 29]. For example, in their investigation of the ISCAS ’85 benchmarks, Hansen et al. analyze replicated structurally isomorphic blocks [9]. The cut-based Boolean matching and aggregation algorithms presented in §2.1 and §2.2 are generalizations of this idea.

A recent attempt at addressing the reverse engineering problem algorithmically is by Li et al. [14]. 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, their 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.

1. Note that 100% identification will not be possible because of the focus on datapath components. This is not a serious limitation as discussed in §5.4 and §6.
An alternative approach to malware detection relies on comparing side channel signals such as power and timing between the trusted design and untrusted versions of the designs. For instance, Agrawal et al. compare “fingerprints” consisting of measurements of power, electromagnetic and thermal emissions [2]. Wang et al. use differences in current measurements to detect trojans [31]. Jin et al. compare path delay measurements [11]. These approaches assume that a trusted (“known good”) version of the chip is available for experimentation, something that may not be true when untrusted component IPs are used, the foundry itself is untrusted or when it is not possible to determine trustworthy chips by destructive examination.

Architectural approaches to trojan detection and avoidance have also been proposed. Hicks et al. [10] proposed an analysis that detects pairs of circuit nodes that are not exercised by design verification tests. They suggested that these nodes could potentially be used to hide trojans and proposed an architectural technique that eliminates such nodes from the circuit and emulates their designed functionality through software. Waksman et al. [30] proposed a set of transformations that permute module inputs, the order in which inputs are applied, and obfuscate reset sequences in order to prevent Trojan activation. Both proposals assume availability of RTL source and design verification tests for the design being analysed.

Trojan detection through algorithmic reverse engineering does not rely on any of these assumptions. Hence, it can detect a wider range of malware, including, for example, bugs/malware introduced by design automation tools. This additional coverage comes at a cost, which is that traditionally reverse engineering has been a labor-intensive process. We show that fully algorithmic reverse engineering is both feasible and effective even for very large designs. A comparison of the differences in assumptions of availability and threat models for the techniques discussed above is shown in Table 1.

This paper builds upon our past work in the area of algorithmic reverse engineering in [26] and [13]. A detailed discussion of the differences between these past efforts and this paper is deferred to Section 6.2.5. The problem of deriving a gate-level netlist from a physical chip is outside the scope of this work. This has been studied in [29, 19, 27, 20, 12]. Nohl et al. derive a gate-level netlist of an RFID tag and examine it for cryptographic vulnerabilities [20]. Kömerling et al. describe techniques for obtaining gate-level descriptions of smartcard processors [12]. Tarnovsky used electron microscopy and bus-level probing to reverse engineer an Secure Infineon Processor which included mesh shielding inserted in order to thwart reverse engineering [27].

1.2 Solution Overview

In this subsection, we describe the assumptions and objectives underlying this work and then provide an overview of our solution.

1.2.1 Assumptions

We tackle the problem of reverse engineering a gate-level netlist under the following assumptions. First, we assume that register transfer language (RTL) source code for the test article being analyzed is not available. We also assume that micro-architectural information as well as design-specific information pertaining to the test article being analyzed is also not available. We only assume availability of “datasheet-level” information which usually consists of a high-level description of the functionality of the test article and a description of its input/output pin interface.

1.2.2 Objective

Given these assumptions our target is to algorithmically derive information about high-level components present in the test article by analyzing the gate-level netlist.

1.2.3 Discussion of Assumptions and Objectives

In both the (a) trojan-detection and (b) intellectual property infringement usage scenarios, our assumptions correspond to an external analyst examining a test article to determine if it (a) contains hardware trojans or (b) infringes relevant intellectual property. Note that the analyst does not have access to source code for the design. Therefore, traditional techniques for trojan detection [2, 31, 11] are not applicable. Furthermore, the only plausible alternative to algorithmically assisted reverse engineering is full manual inspection of the netlist.

In case the assumptions for trojan detection scenario are weaker than ours, we note that our techniques are complementary to the other trojan detection techniques discussed in the previous section. For example, when considering the threat of trojans inserted in the RTL or by design automation tools, analysis techniques like UCI [10] can be synergistically applied along with algorithmic reverse engineering of the gate-level synthesized netlist and correlating identified components with those expected to be present. Note UCI itself cannot detect trojans inserted by design automation tools. Furthermore, all techniques studied in this paper rely on a “static” analysis of the netlist and do not consider information derived from simulations. Combining these algorithms with simulation-based “dynamic” analysis techniques will likely yield interesting results because these approaches are complementary.

Our algorithms focus on identifying datapath components. The are three reasons for this design choice. First, any attack model that is based on triggering malicious behavior through a rare input sequence will necessarily involve some manipulation of the datapath. In fact, as illustrated in §5.4, it is likely that this malicious logic will manifest as a collection of datapath components such as counters, decoders and multiplexers. Thus identifying such components will help an analyst quickly zero-in on problematic parts of the netlist. Second, datapath components exhibit regularity and structure and are amenable to algorithmic analysis. Finally, the majority of the gates in processor-like circuits are in the datapath. The focus on datapath components means that it will not be possible to reverse-engineer 100% of the gates in the design. However, as will be shown in §5.4, this is not a major limitation for detecting hardware trojans.

2. We note that this technique has been defeated [24].

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.
Algorithmic reverse engineering followed by manual analysis.

Statistical comparison of power measurements with good chips.

Hicks et al. [10]

This work.

Statistical comparison of current measures with good chips.

Description

Waksman et al. partial

Identify and eliminate circuit nodes never exercised by verifica-

tion tests.

Statistical comparison of path delays with good chips.

1.2.4 Solution

The objective of our work is to infer a useful high-level description from an unstructured gate-level netlist. In partic-

ular, we focus on reverse engineering datapath elements in
digital circuits. Even when focusing primarily on the datapath,
reverse engineering is still a very hard problem because we are
starting with a sea of gates for the complete chip, including
the datapath as well as the control logic, 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 tackling 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 topologi-

cal/functional analyses. The second stage functionally analyzes potential modules to understand their behavior.

The reverse engineering algorithms introduced by this paper
are as follows:

1) We present a novel application of cut-based Boolean
matching to find replicated bitslices\(^3\) 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 com-
ponents such as multiplexers, adders and subtractors.

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 argu-

ments and results and are matched against a component
library using a Quantified Boolean Formula (QBF) for-

mulation.

5) We present an alternate strategy to infer combinational
modules like decoders using a BDD-based analysis of
nodes with common inputs.

6,7) We present novel algorithms that identify word-level
registers, as well register array structures like register files

<table>
<thead>
<tr>
<th>Detection Technique</th>
<th>Trojan Threat Model</th>
<th>Availability Assumptions</th>
<th>Automation</th>
<th>Paper</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL and test case analysis</td>
<td>Malicious RTL</td>
<td>RTL source, design verification tests</td>
<td>partial</td>
<td>Hicks et al. [10]</td>
<td>Identify and eliminate circuit nodes never exercised by verification tests.</td>
</tr>
<tr>
<td>Algorithmic reverse engineering</td>
<td>Malicious RTL, design tools, foundry</td>
<td>Gate-level netlist</td>
<td>partial</td>
<td>This work.</td>
<td>Algorithmic reverse engineering followed by manual analysis.</td>
</tr>
</tbody>
</table>

---

\(^3\) We define a bitslice as a Boolean function with one output and a small number of inputs that is replicated to construct multibit datapath operators.
and RAM arrays using a BDD-based functional analysis.

8,9) We present algorithms to identify counters and shift registers using topological analyses combined with a satisfiability (SAT) checking formulation.

10) Modules inferred by the above-mentioned portfolio of algorithms may “overlap”, i.e., cover the same elements. These overlaps are resolved by formulating an integer linear program (ILP) that selects a non-overlapping subset of inferred modules that optimizes a set of desired metrics.

### TABLE 2
Netlists Used in Experiments.

<table>
<thead>
<tr>
<th>Design</th>
<th>Chip Inputs</th>
<th>Chip Outputs</th>
<th>Gates</th>
<th>Latches</th>
<th>Flip-flops</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>router</td>
<td>35</td>
<td>26</td>
<td>896</td>
<td>182</td>
<td></td>
<td>on-chip router</td>
</tr>
<tr>
<td>eVoter</td>
<td>31</td>
<td>15</td>
<td>1360</td>
<td>108</td>
<td></td>
<td>voting machine</td>
</tr>
<tr>
<td>cpu8080†</td>
<td>19</td>
<td>26</td>
<td>1807</td>
<td>237</td>
<td></td>
<td>Open CPU</td>
</tr>
<tr>
<td>ae18†</td>
<td>32</td>
<td>64</td>
<td>3466</td>
<td>1094</td>
<td></td>
<td>ae18 CPU</td>
</tr>
<tr>
<td>MIPS16†</td>
<td>1</td>
<td>8</td>
<td>6986</td>
<td>4380</td>
<td></td>
<td>16b MIPS-like core</td>
</tr>
<tr>
<td>oct51†</td>
<td>86</td>
<td>78</td>
<td>8164</td>
<td>2748</td>
<td></td>
<td>8051 microcontroller</td>
</tr>
<tr>
<td>RISC FPU</td>
<td>35</td>
<td>66</td>
<td>14291</td>
<td>3097</td>
<td></td>
<td>RISC FPU</td>
</tr>
</tbody>
</table>

The contributions of this paper include the reverse engineering algorithms listed above as well as the analysis flow shown in Figure 1. We also present a detailed evaluation of our algorithms by experimenting with eight unstructured netlists, details of which are shown in Table 2. The netlists marked with a dagger (†) were obtained by synthesizing designs from opencores.org. Results show that our inference algorithms determine the functionality of more than 45% and up to 93% of gates in the designs in a fully automated manner.

Furthermore, we present a case study of algorithmic reverse engineering of a large highly-optimized system-on-chip (SoC) design consisting of over 375,000 combinational elements. We show a large design like this can be analyzed through logic simplification and module partitioning. Our results show that 68% of the gates left after simplification in this SoC were covered by our inference algorithms. We believe our work is the first effort to algorithmically reverse engineer the majority of the gates in a design of this scale.

A final important contribution of our work is a case study of how our algorithms could aid the detection of hardware trojans. We inject trojans into two test articles and discuss how algorithmic reverse engineering helps a human analyst detect this malicious circuitry.

The rest of this paper is organized as follows. Section 2 describes our algorithms for identifying combinational components. Section 3 describes our algorithms for identifying sequential components. Section 4 describes how possibly conflicting inference results from different algorithms can be resolved to produce the final set of inferred modules. Section 5 presents the experimental evaluation of these algorithms. Section 6 discusses limitations and avenues for further analysis. Section 7 provides some concluding remarks.

### 2 IDENTIFYING COMBINATIONAL MODULES

The section describes algorithms to identify fully combinational modules. Our first algorithm is based on the observation that many datapath elements consist of replicated bitslices connected in a specific topology and is described in §2.1 and §2.2. We then present algorithms for identifying word-level modules in §2.3 and §2.4. Finally, §2.5 presents a third attack on combinational modules using a specific topological property.

#### 2.1 Bitslice Identification

The goal of bitslice identification is to identify all nodes in the circuit that match functions from a bitslice library. For instance, we might be interested in finding all nodes that match the full adder carry function \( f(a, b, c) = ab + bc + ca \), 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 [4, 3].

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 \( G \) [3]. 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. 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 \( f(a, b, c) = ab + c \) and nodes matching \( f(a, b, c) = bc + a \) are grouped into the same class. Each equivalence class may match a known library function.

#### 2.2 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.

##### 2.2.1 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: \( f(a, b, s) = sa + s\overline{b} \). Here we group all matching

---

4. We use the term element to refer to gates, latches and other circuit nodes in the input netlist.

5. These results are in line with published work on cut enumeration [3].
bitslices which have a common select signal (s in this example). Common signal aggregation finds 59 decoders and 140 multiplexers in the RISC FPU test article.

Besides aggregating functions in the bitslice library such as multiplexers and decoders, we can also aggregate unknown functions connected by a common signal to generate candidate unknown modules. These modules may be analyzed either by a human analyst or by a permutation and phase independent matching algorithm such as [18].

2.2.2 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 37 adders/subtractors and 10 parity trees in the RISC FPU test article.

2.3 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.

2.3.1 Symbolic Word Propagation
Once some words are identified, more words can be generated by propagating them across gates. The idea is, given a word w, find conditions under which its value is propagated to a new word w', for every possible value of w.

Consider the circuit in Figure 2. Note that the circuit behaves as a selector between the bitwise negation of the word u1, u2, u3 and the bitwise negation of the word v1, v2, v3 depending on the value of c. Hence, the negated value of u1, u2, u3 gets propagated to w1, w2, w2 if c = 0, and the negated value of u1, u2, u3 gets propagated to w1, w2, w3 otherwise. These are the kind of claims produced by the word propagation algorithm.

For efficiency reasons, we use symbolic simulation, which allows consideration of all possible values of w simultaneously in a single run. Similar to Roth’s D-calculus [22], we redefine functions of logic gates in the circuit operating on the expanded domain \( \{0, 1, D, \bar{D}, X\} \), where D represents a symbolic value in \( \{0, 1\} \), \( \bar{D} \) is the negation of D, and X represents an unknown value. Some examples of symbolic evaluation are: \( \text{and}(D, 1) = D \), \( \text{and}(D, 0) = 0 \), \( \text{and}(0, X) = 0 \), \( \text{not}(X) = X \), and \( \text{not}(D) = \bar{D} \).

Our word propagation algorithm follows a “guess and check” approach. Given an initial word w, the guessing stage consists on finding a set \( S \) of potential “target words” for the propagation. Such set \( S \) is computed by grouping the outputs of the gates driven by the signals in w by gate type and port they connect to. Then, for each target word \( w' \in S \), a set \( C \) of control wires is computed as the set of wires lying in the intersection of the fanins (up to a small depth \( k \)) of the gates whose output is in \( w' \). The checking stage consists on running several symbolic simulations of the local netlist that is relevant to the propagation that is being checked. In such simulations, the inputs of such local netlist are initialized as follows.

- Each bit of \( w \) is set to the symbolic value \( D \).
- For each combination of 3 wires taken from the set of all control wires, all possible binary values are evaluated.
- The rest of the inputs of the local netlist are assigned \( X \).

A simulation with a particular partial assignment \( \sigma \) to the control wires succeeds if all wires in the target word evaluate to either \( D \) or \( \bar{D} \). In that case, \( w \) propagates to \( w' \) under \( \sigma \) and \( w' \) can be tested for further propagation.

An analogous approach in which \( w' \) is guessed among the structural predecessors of \( w \) and it is checked whether \( w' \) can be propagated to \( w \) allows to test for backward propagation.

2.4 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 identifies combinatorial candidate unknown modules operating on words and checks equivalence against a set of predefined reference modules implementing common operations such as addition, subtraction, boolean operations, and shifting/rotation.

For example, consider words \( w_1, w_2, w_3 \), and the largest combinational sub-circuit \( C \) having \( w_1 \) and \( w_2 \) as inputs and \( w \) as output. Additionally, \( C \) may have additional inputs, to which we refer as side inputs. Let \( Y \) be the set of all side inputs of \( C \). Due to optimizations introduced during the synthesis process or simply a design decision, the function implemented by \( C \) might not be unique since the values of some of the wires in \( Y \) determine the operation implemented by \( C \), e.g., addition or subtraction. For this reason, given a reference

---

6. A simple structural analysis is used to find functionally equivalent nodes.
module $C'$ we model our equivalence checking as a 2QBF satisfaction question: is there any value for the wires in $Y$ such that, for every value of the inputs $w_1, w_2$, $C$ and $C'$ give the same output? More concretely, we construct a miter formula $\Phi(X, Y)$ from $C$ and $C'$ by inserting a comparator between their respective outputs. Then, using a state-of-the-art QBF solver, we find values for the side inputs in $Y$ for $C$ to match the function implemented by $C'$.

Fig. 3. QBF formulation showing Miter construction.

The module matching algorithm was able to identify the 8-bit ALU performing addition, subtraction, rotation and negation in the oc8051 test article. Each operation is performed for a different setting of the side inputs, so this module cannot be detected through bitslice aggregation. The ability to create and identify word-level modules was key here.

2.5 Analyses Based on Common Support

In this section we introduce an algorithm that detects modules that do not necessarily have word inputs or outputs or consist of small replicated bitslices. This analysis technique can be used to detect combinational modules with the specific property that each of the outputs of the module depend on the same set of inputs.

Examples of modules which satisfy this property are decoders, demultiplexers and population counters. Note that modules like adders and multipliers do not satisfy this property. Output bit 0 of an adder only depends on the two least significant bits of the addend and the augend, while output bit $k$ of the adder depends on the $k$ least significant bits of the addend and the augend respectively.

2.5.1 Identifying Output Nodes with Identical Supports

Consider the full combinational fanin cone of a combinational node in the circuit. The inputs of this cone are the chip inputs and latch outputs. Suppose two combinational nodes in the circuit are computed using the same set of circuit nodes, it is clear that the inputs of the full combinational fanin cone of these nodes will also be the same.

Therefore, we can group nodes into equivalence classes in the following way. Two nodes are placed in the same class iff the inputs of their full combinational fanin cones are the same. These equivalence classes can be computed efficiently using a union-find data structure and give us candidate output nodes with the property that they are fully determined by the same set of inputs.

Fig. 4. Nodes with common support.

Consider the example shown in Figure 4. Nodes $z_1$ and $z_2$ will be grouped in the same equivalence class because they are completely determined by the same set of chip inputs and latches: $x_1 \ldots x_9$. Such nodes ($z_1$ and $z_2$) will form the outputs of the candidate module.

However, to determine the module boundary we still need to find the inputs of the module, i.e., nodes $y_1$ and $y_2$ in Figure 4. The module boundary is given by the set of nodes in the full combinational fanin cone of the candidate outputs which are not present in the intersection of each of these fanin cones. It is visually clear from the figure that intersection of the combinational fanin cones contains only module 1, so the nodes in the fanin cone which are not present in the intersection leaves us with correct module boundary.

2.5.2 Verifying Module Properties

We use a BDD-based formulation to verify the properties of the modules generated by the algorithm in §2.5.1.

To verify whether a potential module is a decoder or demultiplexer, all that needs to be done is to prove that each output is satisfiable and that no two outputs of the module are simultaneously high. This can be verified in a straightforward manner using a BDD-based analysis.

A population counter can be detected using a similar algorithm which uses BDD-based matching to compare the function of each output node against the function representing each output bit of a population counter.

2.6 Post-Processing of Combinational Modules

Modules generated by the inference algorithms described in this section are subject to a post-processing step that “fuses” certain types of modules to generate larger modules. This increases the level of abstraction of the inferred modules and makes the inference output easier to understand. For example, 2:1 muxes, 3:1 muxes and 4:1 muxes which are adjacent to each other are fused to form larger $n$:1 muxes. Similarly,

8. Assuming the decoder outputs are active-high. The case when the decoders outputs are negated is handled using a symmetric algorithm.

9. Although we verified that the population count algorithm works on artificially constructed circuits with popcnt modules in them, we could not find any population counters in the test circuits we experimented with.
decoders whose outputs drive the select inputs of muxes are fused with the muxes to form routing structures. Module types which can be fused in this manner are said to be compatible.

Module fusion is performed by first constructing a module fusion graph. The nodes in the graph are modules and an edge between Module A and Module B exists in the graph if and only if all the outputs of module A are inputs of module B and respective module type are compatible. Once the module fusion graph is constructed, connected components in the graph are fused to form a larger combinational module and the resulting module is added to collection of inferred modules. The constituent modules which were the “inputs” of the fusion are not eliminated at this stage. The overlap elimination algorithm (see §4) determines which of these modules (fused vs. constituents) is included in the output.

3 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, shift registers and multibit registers.

3.1 Counter Identification

The specific problem in counter identification is to identify sets of latches in the unstructured netlist that behave like counters. The difficulty here is twofold. First, given a set of latches that we suspect to implement a counter, we need a functional analysis that can verify its properties. Second, we need an efficient algorithm to enumerate candidate counters. Simply considering all subsets of latches is computationally infeasible.

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 shown in Figure 5. The intuition for this topology is that bit $i$ of an $n$-bit counter toggles when all the lower order bits $1 \ldots i - 1$ are all high. Therefore, there needs to be a combinational path from the outputs of these latches to the input of bit $i$, leading to the topology shown in the figure.

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.

3.1.1 Topological Check Using the LCG

The latch connection graph (LCG) is an unweighted graph $G = (V, E)$ which formalizes the notion of information flow between latches. The vertices of the graph $V$ are the latches and flip-flops in the netlist being analyzed. A directed edge $(v_1, v_2) \in E$ if there is a combinational path from the output of latch $v_1$ (its Q node) to the input of latch $v_2$ (its D node).

![Fig. 5. Latch-to-latch information flow in a counter: each latch in the counter is driven by the latches corresponding to the lower-order bits.](image)

Given the LCG, we find subgraphs which have the topology shown in Figure 5. More precisely, given the LCG $G = (V, E)$, we find ordered sets of nodes $V_c = \{v_1, \ldots, v_k\}$, such that $V_c \subset V$ and $\forall v_i, v_j \in V_c : (v_i, v_j) \in E$ iff $i \leq j$.

3.1.2 Verifying Counter Properties

We now devise a functional analysis that verifies that the “candidate” counter found by the topological analysis has the properties of a counter. First, let us formalize the behavior of an up counter as follows.

$$
c_i = \neg r \land e \land (q_1 \land q_2 \land \cdots \land q_{i-1}) \land \neg q_i \lor
\neg r \land e \land (\neg q_1 \lor \neg q_2 \lor \cdots \lor \neg q_{i-1}) \land q_i \lor
\neg r \land e \land q_i \lor s
\tag{1}
$$

In the equation above, $c_i$ determines the next state of bit $i$ of an $n$-bit counter. $r$ is the function that resets the counter, $s$ is the function that sets its value high, $e$ is the count-enable function, and $q_1 \ldots q_i$ are the current values of latches $1 \ldots i$ of the counter.

Equation (1) says that bit $i$ toggles when all the lower order bits ($1 \ldots i - 1$) are high, the counter is enabled and not being reset. The bit retains its value when the counter is enabled, not reset but one of the lower order bits is zero. The counter also holds its value when it is not reset and not enabled. Bit $i$ is pulled high if the set function evaluates to 1. Note since we have left the functions $r$, $e$ and $s$ unspecified, $c_i$ is actually a family of functions and not a specific Boolean function.

Now consider the Boolean function defined by the full combinational fanin cone for each latch in the candidate counter. Let this function be denoted by $d_i$ where $i$ ranges

---

10. For clarity of presentation the rest of this section focuses on up counters. Our implementation uses symmetric techniques to detect down counters.
across the bits of the counter. We compute the following cofactors from $d_i$:

$$f_i = \text{cofactor}(d, q_i \land q_{i-1} \land \ldots \land q_1)$$
$$g_i = \text{cofactor}(d, q_i \land q_{i-1} \land \ldots \land q_1)$$
$$h_i = \text{cofactor}(d, q_{i-1} \land \ldots \land q_1)$$

The insight here is that if the function $d_i$ is compliant with $c_i$ from Equation (1), then the functions $f_i$, $g_i$ and $h_i$ will reduce to Equation (2).

$$f_i = (\neg r \land e) \lor s$$
$$g_i = (\neg r \land \neg e) \lor s$$
$$h_i = \neg r \lor s$$

Now, the functions $r$, $s$ and $e$ should be the same for all the bits in the counter. Hence, $f_i$, $g_i$ and $h_i$ must also be equivalent. Therefore, we can determine that a set of latches is not a counter if the SAT solver finds that the functions $f_i$, $g_i$ and $h_i$ are not equivalent for all $i$.

Five counters were found in the oc8051 test article.

### 3.2 Shift Register Identification

As with counters, our goal here is to identify sets of latches 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 except that the topology and verification conditions differ.

#### 3.2.1 Topological Check

The topological check for shift registers uses a pruned version of the latch connection graph (LCG) that we call the single path latch connection graph (SPLCG). As in the LCG, the nodes in the SPLCG are the latches and flip-flops in the netlist. However, the edge $v_1 \rightarrow v_2$ exists in the SPLCG iff there is exactly one combinational path from the output of latch $v_1$ to the input of latch $v_2$.

The topological check for unidirectional shift registers is as follows. Given the SPLCG $G = (V, E)$, we find ordered sets of nodes $V_s = \{v_1, v_2, \ldots, v_k\}$ such that $V_s \subset V$ and $\forall v_i, v_j \in V_s : (v_i, v_j) \in E$ iff $j = i + 1$. In other words, we are searching for chains of latches connected by exactly one combinational path between each latch and its successor.

#### 3.2.2 Verifying Shift Register Properties

We model the family of functions representing the next-state function of bit $i$ of a shift register using the following equation.

$$s_i = \neg r \land (e \land q_{i-1} \land \neg e \land q_1) \lor s$$

$r$, $s$ and $e$ are the reset, set and enable functions respectively. $q_i$ is the output of the $i$th latch of the shift register. Suppose $d_i$ is the Boolean function determined by the full combinational

fan-in cone of latch $i$ of the supposed shift register. As in the counter analysis, we consider the following cofactors of $d_i$.

$$f_i = \text{cofactor}(d, q_{i-1} \land \neg q_i)$$
$$g_i = \text{cofactor}(d, \neg q_{i-1} \land q_i)$$

If $d_i$ is compliant with $s_i$ we will have:

$$f_i = \neg r \land e \lor s$$
$$g_i = \neg r \land \neg e \lor s$$

Therefore, the functional check verifies that the functions $f_i$ and $g_i$ are identical for each bit of the shift register.

#### 3.2.3 Shift Register Aggregation

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.

### 3.3 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”).

#### 3.3.1 Identifying Read Logic

The intuition behind identifying the read-logic is that it forms a set of trees with the RAM cells as leaves of the tree and the read outputs as roots. We use a marking algorithm to find such trees. Initially, the algorithm marks all latches in the netlist. Subsequently, it marks all gates which satisfy the following conditions: (i) at least one of the gate’s inputs is marked and (ii) the gate has only one fanout. This is repeated until no new nodes are marked.

#### 3.3.2 Verification of Read Behavior

The next step is a functional analysis of the marked nodes. A BDD is constructed for each marked node in terms of the latches, inputs and unmarked nodes in the circuit. Among the inputs of this BDD, we assume that those which are latches are storage nodes ($l_i$) while the remaining are the read address ($s_i$). We then verify the following properties.

1. If $y = f(s_1, s_2, \ldots, s_k, l_1, \ldots, l_n)$ then $y = l_i$ or $y = \neg l_i$ for every value of $s_1 \ldots s_k$. In other words, each select input propagates exactly one of the latches to the output.
2. Every latch node is propagated to the output: $y = l_i$ or $y = \neg l_i$ for all $i$ and appropriate $s_1 \ldots s_k$.

Latches and nodes which pass these checks are identified as the RAM array and its corresponding read-logic.12

11. Given Boolean functions $f$ and $g$, $\text{cofactor}(f, g)$ is the function obtained when $f$ is evaluated over the restricted domain specified by $g = 1$.

12. The analysis handles each bit output of the array independently so sets of latches with common select inputs (read addresses) are aggregated to form an array with multibit inputs/outputs.
3.3.3 Identifying Write Logic

Fig. 6. RAM write-logic: \( w_e_i \) is the write-enable signal for word \( i \) and \( wd_i \) is the data to be written to word \( i \).

The logic that controls RAM writes is shown in Figure 6. It consists of decoders driving 2:1 muxes that select between the write-data and the latch output. The muxes drive the latch inputs and their select signal is the write-enable signal, denoted by \( w_e_i \). Once the latches that comprise the register file are known, cut matching can give us these muxes. Our algorithm then computes the BDDs for each write-enable signal using the intersection of combinational fan-in cones. The following properties are then verified: 13

1) Each write-enable signal is satisfiable: \( w_e_i \neq 0 \).
2) No two write-enable signals are simultaneously satisfiable: \( w_e_i \land w_e_j = 0 \) if \( i \neq j \).

If these properties are satisfied, the set of gates that comprise the latch inputs, muxes and common support nodes are identified as the write-logic.14

One RAM structure, a 32x32b register file with two read ports and 1 write port, was detected in the RISC FPU.

3.4 Identifying Multibit Registers

We use the term multibit register to denote a set of 1-bit registers whose values are updated in tandem.

Each cycle either one of three different values: \( v[1] \), \( v[2] \), or \( v[3] \) is assigned to the register based on the conditions \( c_1 \), \( c_2 \) and \( c_3 \). A structure of this form can be detected using bitslice matching and aggregation to find the multibit multiplexer and then examining the fanouts and inputs of the multiplexer to detect the sequential elements around it.15

39 multibit register elements were found in the RISC FPU.

4 OVERLAP RESOLUTION

The inference algorithms described in this paper operate independently. Therefore, it is possible that a particular gate in the netlist under analysis might be placed in multiple inferred modules. For example, in the oc8051 design, the RAM read-array consists of many muxes identified by the bitslice aggregation algorithms and the RAM analysis algorithm.

One idea would be to output all inferred modules and allow a human analyst to pick and choose the “correct” non-overlapping description of the circuit. While this may be a feasible option for small circuits, for some of the larger circuits, the inference tool produces several tens of thousands of modules. It would be infeasible for a human to look through all these modules and select a non-overlapping subset.

In this section, we investigate algorithmic techniques for generating a non-overlapping subset of inferred modules given the output of the portfolio. In particular, we would like to generate non-overlapping subsets that either (i) maximize coverage (measured by number of gates identified) or (ii) minimize the number of inferred modules while meeting a coverage target. The former objective is desirable because it attempts to identify as many gates as possible. The latter is interesting because we expect that an inference output with fewer modules while meeting the required coverage target would be easier to understand from a human analysts’ point of view.

4.1 Basic Formulation Overview

At a high-level, our solution involves formulating a binary integer linear program (BILP; sometimes called a Zero-One ILP) that selects a non-overlapping subset of modules that optimizes for the desired target metric. We describe the formulation of the ILP in the following subsection.

4.1.1 ILP Variables

The basic formulation requires one binary variable for each inferred module. Suppose there are a total of \( M \) modules, then the formulation has \( M \) binary variables \( x_1, x_2, x_3, \ldots, x_M \). Setting variable \( x_i = 1 \) denotes that module \( i \) will be selected for output, while \( x_i = 0 \) means that module \( i \) will be elided from the output.

4.1.2 Constraints Describing Overlaps

Consider an arbitrary element \( g_k \) from the netlist being analyzed. Suppose this element \( g_k \) is covered by inferred modules \( k_1, k_2, \ldots, k_l \). To represent the requirement that element \( g_k \)

13. This presentation assumes the write-enable signal is active high, but it could also be active-low in which case the properties are modified appropriately. We determine the polarity of the write-enable signal by examining which of the mux inputs is connected to the latch output.
14. We note that the analysis is unable to determine the ordering of the bits in inputs and outputs of the RAM.

15. As in the case of the RAM identification, the analysis is unable to determine the ordering of the bits in the multibit register. In some cases, we were able to infer the ordering of bits by seeding the symbolic word propagation algorithm with ordered words and checking whether one of the propagated words matched the register outputs. Note ordered words can be inferred from aggregation algorithms for adders and subtractors.
can be covered by only one of these modules in the final output, we introduce a constraint of the following form.

\[ x_{k_1} + x_{k_2} + \cdots + x_{k_l} \leq 1 \]

There are as many constraints as there are elements in the netlist that are covered by multiple modules.

### 4.1.3 Objective Function

The objective of maximizing coverage is encoded in a straightforward manner. Let the "size" (i.e., the number of elements covered by) module \( i \) be \( S_i \). Then the objective function is:

\[ \text{maximize } \sum_{i=1}^{M} x_i \cdot S_i \]

### 4.1.4 Alternative Formulation: Minimize Inferred Modules Given Coverage Constraint

This formulation minimizes the number of output modules while introducing a new constraint that ensures that a certain coverage target is met. We retain the same variables as the previous formulation (described in §4.1.1) and use the same constraints to encode the selection of non-overlapping modules (see §4.1.2). The objective is as follows.

\[ \text{minimize } \sum_{i=1}^{M} x_i \]

We also need to introduce a new constraint that encodes the fact that the coverage target of \( C_t \) elements must be met. This is done by adding the following constraint to the ILP.

\[ \sum_{i=1}^{M} x_i \cdot S_i \geq C_t \]

### 4.2 Sliceable Formulation

Inferred modules are grouped into two categories. Modules like muxes and decoders which can be split up into independent bitslices are considered "sliceable". If such a module has \( n \) slices, it is modelled in the ILP with \( n+1 \) binary variables: \( x_{i_0}, x_{i_1}, \ldots, x_{i_n} \). Variable \( x_{i_j} \) where \( j \geq 1 \) represents whether slice \( j \) of module \( i \) is selected for output. Variable \( x_{i_0} \) is a special variable introduced for technical reasons. It represents where module \( i \) itself (i.e., any slice in module \( i \)) is selected for output. In the example shown in Figure 8, suppose the 5-bit multiplexer is module \( i \), then MUX bitslice 1, 2, 3, 4 and 5 will be represented by variables \( x_{i_1}, x_{i_2}, x_{i_3}, x_{i_4} \) and \( x_{i_5} \) respectively.

Modules which are not "sliceable", for example: counters and RAMs, are represented as before with a single binary variable \( x_i \) which determines whether the entire module is selected for output or discarded.

#### 4.2.2 Constraints Describing Overlaps

The formulation in §4.1.2 expressed the fact that if a gate is covered by \( l \) different modules, no more than one of these modules could be selected for output. Here, we would like to express the same but at the finer granularity of slices rather than modules. For this it is necessary to assign the elements included in a module to its component slices.

Define the following function \( \text{Var}(g_k, i) \) that maps an element \( g_k \) contained in module \( i \) to the ILP variable that represents the slice that \( g_k \) is contained in.

\[ \text{Var}(g_k, i) = \begin{cases} x_i & \text{if module } i \text{ is unsliceable} \\ x_{i_j} & \text{if } g_k \text{ is contained only in slice } j \text{ of module } i \\ x_{i_0} & \text{otherwise} \end{cases} \]

The intuition here is that for a sliceable module, elements which are contained in exactly one slice are mapped to that slice. Elements which are contained in more than one slice, are mapped to the variable \( x_{i_0} \) which is the special variable that represents the entire module. Returning to the example in Figure 8, the gates which are inside the boxes labelled “MUX bitslice \( j \)” will be mapped to variable \( x_{i_j} \). The inverter however, is “part of” all bitslices, so it is mapped to \( x_{i_0} \).

As before, suppose element \( g_k \) is covered by inferred modules \( k_1, k_2, \ldots, k_l \). We add the following constraint.
\[ \text{Var}(g_k, k_1) + \text{Var}(g_k, k_2) + \cdots + \text{Var}(g_k, k_l) \leq 1 \]

Consider a gate that is contained within the box labelled “MUX Bitslice 4” in Figure 8. The specific constraint introduced by a gate inside “MUX Bitslice 4” will be \( x_{i_4} + x_j \leq 1 \). This tells the solver that either bitslice 4 or the RAM can be selected for output but not both. Unlike in the basic formulation, we are not restricting the selection of the other bitslices in the MUX.

### 4.2.3 Slice-Related Constraints

For each sliceable module, we would like to specify that if any individual slice is selected, gates that are common to more than one slice are also selected. This leads to constraints of the following form.

\[
\text{forall } 1 \leq j \leq n: x_{i_0} - x_j \geq 0
\]

In the notation above, module \( i \) has \( n \) slices and is modelled in the ILP using the variables \( x_{i_0}, x_{i_1}, \ldots, x_{i_n} \).

We would also like to specify that each module contains a minimum number of slices to avoid creating very small modules. This is done using a constraint of the form:

\[
\sum_{j=1}^{n} x_{i_j} - \text{MinSlices} \cdot x_{i_0} \geq 0
\]

\( n \) is the number of slices in module \( i \). \(^{16} \) All results shown in this paper are with \( \text{MinSlices} = 2 \).

### 4.2.4 Objective Function

The objective function to maximize coverage is similar to that presented in §4.1.3 with the difference that we have to count “sizes” on a per-slice basis. Define the size function as follows.

\[ \text{Size}(x) = \left\{ \{ g_k \mid \text{Var}(g_k, i) = x \text{ for some } i \} \right\} \]

Clearly, \( \text{Size}(x) \) counts the number of elements covered by the variable \( x \). Given \( \text{Size}(x) \) the objective function can be derived in a straightforward manner by weighting each variable with its corresponding size.

\[ \text{maximize } \sum_{\text{variable } x} x \cdot \text{Size}(x) \]

Returning to the example in Figure 8, the solver can maximize coverage by setting \( x_{i_0}, x_{i_1}, x_{i_2}, x_{i_3} \) and \( x_j \) to 1 and \( x_{i_4} \) and \( x_{i_0} \) to zero. This satisfies all the constraints we have described and selects bitslices 1, 2 and 3 of the MUX and the entire RAM.

\(^{16} \) Note adding the constraint \( \sum_{j=1}^{N} x_{i_j} \geq \text{MinSlices} \) is incorrect. This requires every module to have \( \text{MinSlices} \) slices selected. What we want is: if a module is selected, it must have at least \( \text{MinSlices} \) slices in it.

### 4.2.5 Alternative Formulation

The formulation that minimizes the number of inferred modules while meeting the coverage target \( C_t \) again requires the addition of the following constraint.

\[ \sum_{\text{variable } x} x \cdot \text{Size}(x) \geq C_t \]

The following function returns the representative variable for a module \( i \). The representative variable determines whether a module is selected for output.

\[ \text{rep}(i) = \begin{cases} x_{i_0} & \text{if module } i \text{ is not sliceable} \\ x_{i_0} & \text{if module } i \text{ is sliceable} \end{cases} \]

In the example shown in Figure 8, the representative variables for the 5-bit multiplexer and RAM are \( x_{i_0} \) and \( x_j \) respectively.

Assuming the total number of modules is \( M \) and that they are numbered from 1 to \( M \), the objective function is now given by the following equation.

\[ \text{minimize } \sum_{i=1}^{M} \text{rep}(i) \]

## 5 Experimental Results

We now present a detailed evaluation of our algorithms.

### 5.1 Methodology

We developed an inference tool using the C++ and Python programming languages that implement the algorithms described in this paper. The tool takes as input a synthesized Verilog netlist, analyzes it and outputs an abstracted netlist with the inferred components. The tool uses the CU Decision Diagram (CUD) Package version 2.4.2 for the BDD-based analyses \([23]\), and MiniSat version 2.2 for satisfiability checking \([6]\). DepQBF \([16]\) was used as the QBF solver and IBM CPLEX version 12.5 was the ILP-solver.

Experiments were performed on an Intel® Xeon® E31230 CPU clocked at 3.20GHz with 32 GB of RAM. One set of results are shown for eight netlists. Details of these netlists are shown in Table 2. All the designs were synthesized using an IBM/ARM cell library for a 45nm SOI process. This paper also shows inference results on a large highly-optimized SoC design consisting of more than 375,000 combinational elements. A case study describing our analysis of this test article is given in Section 5.3. Finally, we describe a case study where we inject hardware trojans into two of the test articles from Table 2 and discuss how our algorithms would aid an analyst detect these trojans.

### 5.2 Summary of Results

Table 3 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

16. Note adding the constraint \( \sum_{j=1}^{N} x_{i_j} \geq \text{MinSlices} \) is incorrect. This requires every module to have \( \text{MinSlices} \) slices selected. What we want is: if a module is selected, it must have at least \( \text{MinSlices} \) slices in it.
netlists being analyzed, the number of inferred modules of various types and the execution time of the tool.

For each test article, we show two rows. The white row shows the number of modules obtained before overlap resolution. This means that for the results shown in the white rows, each gate/flip-flop/latch in the test article may be placed into multiple different inferred modules. These results are directly comparable to the results presented in [26]. The shaded rows show the results after overlap resolution (§4) has been performed. In this case, each gate/latch/flip-flop is placed in at most one inferred module. The process of overlap resolution necessarily involves a small loss in coverage but we see from the results shown that the loss is quite small.

For the three biggest netlists, coverage is above 70% 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.

Both the execution time and memory requirements posed by the analysis tool are very reasonable. The maximum execution time among this set of designs is a little more than three minutes and the maximum resident set size is 4.1GB. The most computationally-expensive algorithm in our toolbox is the counter analysis.

The white rows show results before overlap resolution while the shaded rows are inferred modules obtained after overlap resolution.

### TABLE 3
Coverage Results.

<table>
<thead>
<tr>
<th>Design Information</th>
<th>Combinational components</th>
<th>Sequential components</th>
<th>Coverage and execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>gate</td>
<td>latch</td>
<td>a/s</td>
</tr>
<tr>
<td>router</td>
<td>896</td>
<td>182</td>
<td>0</td>
</tr>
<tr>
<td>eVoter</td>
<td>1360</td>
<td>108</td>
<td>0</td>
</tr>
<tr>
<td>Open8</td>
<td>1807</td>
<td>237</td>
<td>0</td>
</tr>
<tr>
<td>cpu8080</td>
<td>2258</td>
<td>243</td>
<td>0</td>
</tr>
<tr>
<td>ae18</td>
<td>3466</td>
<td>1094</td>
<td>0</td>
</tr>
<tr>
<td>MIPS16</td>
<td>6986</td>
<td>4380</td>
<td>0</td>
</tr>
<tr>
<td>oc8051</td>
<td>8164</td>
<td>2748</td>
<td>0</td>
</tr>
<tr>
<td>RISC FPU</td>
<td>14291</td>
<td>3097</td>
<td>0</td>
</tr>
</tbody>
</table>

Legend for table header: gate: number of gates; latch: number of latches a/s: adders/subtractors; 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; reg: registers; cov: coverage in percentage of gates covered; tim: execution time in seconds. mem: maximum resident set size in GB (proxy for memory consumption).

The white rows show results before overlap resolution while the shaded rows are inferred modules obtained after overlap resolution.

### 5.3 Case Study 1: Analysis of BigSoC Test Article

We present a case study of algorithmic reverse engineering of large, realistic SoC. This SoC consists of 375090 combinational elements, 34318 latches and 62 and 94 inputs and outputs respectively. We describe the three part strategy we used to analyze this design in the rest of this subsection.17 Besides the gate-level netlist, we were also given a datasheet.

17. The generic name “BigSoC” is used for confidentiality reasons.
for the SoC. The datasheet listed the seven constituent cores of the SoC and provided a brief description of the high-level functionality of each core.

5.3.1 Circuit Simplification

The SoC in its raw form contains many redundant combinatorial elements such as delays, buffers and paired inverters which were inserted presumably for electrical reasons. These elements result in the inference of many functionally equivalent modules with slightly different module boundaries and adversely affect computational performance and scalability. Therefore, our first step was to perform structural logic simplification and eliminate buffers, delays, paired inverters and a few other structurally equivalent gates. This reduced the number of combinational elements in the SoC from 375090 to 168730, a reduction of about 55%!

5.3.2 Partitioning by Reset Tree

Even after logic simplification, we found that the computationally expensive analysis algorithms - counter and shift register detection - timed out on the complete design. Although these inferred modules are small, they are important in gaining insight into the working of the design. Therefore, the second step in our analysis of the SoC was to improve analysis scalability by partitioning the SoC into its constituent cores.

The datasheet of the SoC informed us that the SoC had seven constituent cores. The SoC had individual reset inputs for each of these cores and we used these inputs in partitioning the SoC into its constituent parts. The partitioning algorithm marks each latch with all the reset inputs that are in its combinational fan-in cone. The union of the set of all latches marked with a module’s reset input and all the gates in their respective fan-in cones yields the module partitioning.

TABLE 5

BigSoC Partition Information.

<table>
<thead>
<tr>
<th>Partition</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Gates</th>
<th>Latches</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C</td>
<td>472</td>
<td>432</td>
<td>2027</td>
<td>416</td>
<td>PC impl.</td>
</tr>
<tr>
<td>MemCtrl</td>
<td>617</td>
<td>568</td>
<td>3262</td>
<td>571</td>
<td>mem. controller</td>
</tr>
<tr>
<td>SPI</td>
<td>470</td>
<td>429</td>
<td>1980</td>
<td>415</td>
<td>SPI interface</td>
</tr>
<tr>
<td>UART</td>
<td>463</td>
<td>397</td>
<td>3052</td>
<td>656</td>
<td>UART core</td>
</tr>
<tr>
<td>VGA</td>
<td>477</td>
<td>621</td>
<td>19422</td>
<td>770</td>
<td>VGA core</td>
</tr>
<tr>
<td>ARM</td>
<td>786</td>
<td>605</td>
<td>22122</td>
<td>4316</td>
<td>32b ARM</td>
</tr>
<tr>
<td>SVD</td>
<td>1061</td>
<td>1044</td>
<td>100205</td>
<td>20419</td>
<td>SVD core</td>
</tr>
<tr>
<td>BigSoC</td>
<td>62</td>
<td>94</td>
<td>168730</td>
<td>34318</td>
<td>full simplified SoC</td>
</tr>
</tbody>
</table>

The details of the partitioning are shown in Table 5. Note that a very small number of gates (176 or 0.1%) of gates are placed into more than one module. We assert that this discrepancy can easily be resolved by a human analyst during a later stage of the investigation. About 5% of the gates are not placed in any partition. We believe these gates correspond to an inter-core interconnect mentioned in the datasheet.

5.3.3 Results for BigSoC

Results of analyzing the partitions as well as the entire SOC are shown in Table 6. We note that the coverage is between 62% and 88%. The VGA module contains a 12000+ gate “framebuffer read” structure that was detected using a design-specific algorithm. These results demonstrate that our inference algorithms are effective on very realistic large SoC designs. The computational requirements for the analysis are reasonable and the entire analysis can be performed in slightly over two hours on a contemporary midrange server CPU.

5.4 Case Study 2: Trojan Detection

In order to demonstrate how our reverse engineering algorithms can aid trojan detection, we now present a case study where we analyze trojan-injected versions of two of the test articles studied above. Our goal here is demonstrate how inferences from our analysis algorithms can aid a human analyst who is trying to detect the presence of hardware trojans in a gate-level netlist. As stated previously, we assume that the analyst does not have access to the RTL source code and/or known good chips and has to rely on manual and algorithmic analysis of the gate-level netlist to detect malicious behavior.

5.4.1 Description of Trojans

We injected trojans into the oc8051 and eVoter test articles. A comparison of the original and the trojan-inserted versions of these test articles is shown in Table 7.

In the case of the eVoter, the trojan is activated by a secret seven key sequence and allows selection of a specific candidate. All subsequent votes now go to this candidate. The trojan can be deactivated by pressing the secret key sequence again. In this case, the trojan is a backdoor that can be used to compromise the voting machine hardware.

In the case of the oc8051, the trojan circuitry is activated when an XOR instruction is a repeated 5 times in a row. Once the trojan is activated all outputs from the ALU to the accumulator are set to zero. In other words, the trojan here is a kill-switch activated by a rare sequence of instructions.

TABLE 7

Details of Trojan-Inserted Designs.

<table>
<thead>
<tr>
<th>Design</th>
<th>Chip Inputs</th>
<th>Chip Outputs</th>
<th>Gates</th>
<th>Latches</th>
<th>Flip-flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>eVoter</td>
<td>31</td>
<td>15</td>
<td>1360</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>eVoter w/ trojan</td>
<td>31</td>
<td>15</td>
<td>1416</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td>oc8051</td>
<td>86</td>
<td>78</td>
<td>8164</td>
<td>2748</td>
<td></td>
</tr>
<tr>
<td>oc8051 w/ trojan</td>
<td>86</td>
<td>78</td>
<td>8189</td>
<td>2759</td>
<td></td>
</tr>
</tbody>
</table>

5.4.2 Results of Algorithmic Inference

Table 8 summarizes the results of the inference algorithms on the trojan-inserted designs. We chose to show the results before and after overlap resolution because the resolution algorithm may discard modules that provide insight into the trojan because these modules overlap with other inferred modules.

5.4.3 eVoter Trojan Analysis

In the case of the eVoter trojan, we see that several additional decoders and demultiplexers are inferred. These modules correspond to the logic in the trojan that matches the specific secret key sequence which activates the trojan. Further, we see two additional muxes and one additional multibit register. The mux and multibit register here are especially prominent
because there are only a few of these modules in the design. In fact, the additional modules here precisely correspond to the logic that overrides the user input button number (i.e., user vote) with the secret trojan/backdoor input.

A human analyst analyzing the inferred modules with no prior knowledge of the trojan is likely to have noticed this mux that selects either the user input button number or a multibit register. Further, the analyst would have noticed several decoders, which are part of a state machine, and that these decoders are also driven by the input button number. Combining this with some manual analysis of the state machine and discovering that the state machine drives the select input of this mux would very likely have led to the discovery of the hardware trojan.

5.4.4 oc8051 Trojan Analysis
The most important additional module discovered by the algorithms for the oc8051 trojan is a counter. This is in fact the counter which counts the number of consecutive XOR instructions being executed by the ALU. The tool also discovered a gating function module that zeros out the ALU. XOR instructions being executed by the ALU. The tool also discovered a gating function module that zeros out the ALU.

6 Discussion
In this section, we discuss some of the limitations and areas for potential improvement in our tools. We also provide a detailed comparison with our previous work which this paper builds on.

6.1 Abstraction Quality
Some of the inferred modules detected by the tool, such as decoders and demultiplexers are somewhat small and cover tens of gates leading to a moderately large number of such modules in the output. Due to this, a human analyst would need to spend more time looking at each of these inferred modules. At first sight, this appears to be a major limitation. However, it is important to note that even with these small modules, the number of inferred modules is at least an order magnitude and usually a few orders of magnitude fewer than

| TABLE 6 |
| Coverage Results on BigSoC Partitions. |

<table>
<thead>
<tr>
<th>Design</th>
<th>gate</th>
<th>latch</th>
<th>a/s</th>
<th>dec</th>
<th>dm</th>
<th>eq</th>
<th>gf</th>
<th>mux</th>
<th>h</th>
<th>ram</th>
<th>sr</th>
<th>cnt</th>
<th>reg</th>
<th>misc</th>
<th>cov</th>
<th>tm</th>
<th>mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2C</td>
<td>2027</td>
<td>416</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>26</td>
<td>68</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>62%</td>
<td>16s</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>MemCtrl</td>
<td>3262</td>
<td>571</td>
<td>1</td>
<td>15</td>
<td>1</td>
<td>45</td>
<td>41</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>63%</td>
<td>33s</td>
<td>1.6</td>
<td></td>
</tr>
<tr>
<td>SPI</td>
<td>1980</td>
<td>415</td>
<td>1</td>
<td>15</td>
<td>0</td>
<td>23</td>
<td>47</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>64%</td>
<td>15s</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>UART</td>
<td>3052</td>
<td>656</td>
<td>1</td>
<td>10</td>
<td>2</td>
<td>47</td>
<td>67</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>25</td>
<td>0</td>
<td>64%</td>
<td>21s</td>
<td>1.2</td>
<td></td>
</tr>
<tr>
<td>VGA</td>
<td>19422</td>
<td>770</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>404</td>
<td>219</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>1</td>
<td>88%</td>
<td>1545s</td>
<td>2.8</td>
<td></td>
</tr>
<tr>
<td>ARM</td>
<td>22122</td>
<td>4316</td>
<td>2</td>
<td>101</td>
<td>1</td>
<td>301</td>
<td>783</td>
<td>21</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>35</td>
<td>0</td>
<td>62%</td>
<td>298s</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>SVD</td>
<td>109205</td>
<td>20419</td>
<td>380</td>
<td>295</td>
<td>4</td>
<td>1207</td>
<td>956</td>
<td>225</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td>56</td>
<td>0</td>
<td>70%</td>
<td>1721s</td>
<td>9.1</td>
<td></td>
</tr>
<tr>
<td>BigSoC</td>
<td>168730</td>
<td>34318</td>
<td>389</td>
<td>605</td>
<td>21</td>
<td>1263</td>
<td>2102</td>
<td>181</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>283</td>
<td>3</td>
<td>68%</td>
<td>6029s</td>
<td>17.1</td>
<td></td>
</tr>
</tbody>
</table>

Note: Table header legend is the same as for Table 3. Entries with dashes indicate algorithms which timed out on that particular netlist.

Miscellaneous components include two clock tree modules and one frame buffer read structure in the VGA core.

| TABLE 8 |
| Trojan Analysis Results And Comparison. |

<table>
<thead>
<tr>
<th>Design Information</th>
<th>Combinational components</th>
<th>Sequential components</th>
<th>Coverage and execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>gate</td>
<td>latch</td>
<td>a/s</td>
</tr>
<tr>
<td>eVoter</td>
<td>1360</td>
<td>108</td>
<td>0</td>
</tr>
<tr>
<td>eVoter w/ trojan</td>
<td>1416</td>
<td>117</td>
<td>0</td>
</tr>
<tr>
<td>oc8051</td>
<td>8164</td>
<td>2748</td>
<td>20</td>
</tr>
<tr>
<td>oc8051 w/ trojan</td>
<td>8189</td>
<td>2759</td>
<td>18</td>
</tr>
</tbody>
</table>

Note: Table header legend is the same as for Table 3. Rows shaded gray show inferred modules after overlap resolution while the unshaded rows show the inferred modules before overlap resolution. Columns which are in bold show additional inferred components which correspond to the inserted trojan.
the number of gates and registers present in the article being analyzed. As we pointed out in the discussion of the trojan injected in oc8051, this results in a very significant reduction in the workload for an analyst.

Furthermore, if design-specific information is available about the types of modules expected to be present in the design, our algorithms can be easily extended to detect such modules. Two examples are shown in this paper. The first is the framebuffer-read structure in the VGA module of BigSoC. In this case, we designed the algorithm to detect this structure knowing that a VGA controller and framebuffer were present in the design.

A second example is our detection of the ALU in oc8051. Detecting an ALU requires knowledge of the exact functions implemented by an ALU such as add, subtract, not, negate, and, nand, or, xor, xnor etc. These vary from design to design. Moreover, the number of ALU operation specifier bits, the number of input operands and their bitwidths also needs to be known or derived. These factors make it hard to write a completely general ALU detection algorithm. However, in the case of the oc8051, analyzing the instruction set gave us information about the functions performed by the ALU as well as the width of each ALU input. In this case, we were able to use QBF-based module matching and word identification to precisely identify the ALU in the oc8051 design. Building such a large library of high-level components is an important topic for future work.

Finally, we wish to point out that it is actually advantageous for the tool to detect small “building-block” type of modules in the trojan detection scenario. Small modules like decoders, counters, multiplexers and gating functions are the building blocks from which higher-level functionality is derived. Since trojans can be implemented in a variety of different ways, detecting these building blocks is a more promising approach than designing algorithms that detect high-level modules that correspond to trojans. Such algorithms will likely be stymied by small differences in implementation of the trojan.

6.2 Improving Coverage

Our algorithmic inference tool can automatically reverse engineer between 45% and 93% of the gates in chip. This still leaves a significant number of gates that need to be reverse engineered to completely understand the chip’s functionality. In the rest of the section, we discuss some ways of reverse engineering these gates.

It is important to note that placing 100% of the gates and registers in a design into inferred modules is not necessary for trojan detection. As we showed in §5.4, identifying a few key modules in the trojan is sufficient to alert an analyst to potentially malicious behavior. And because our algorithms can infer a rich library of logical building blocks, we assert that a significant part of almost any trojan would be covered by the algorithms presented here.

6.2.1 Design-Specific Bitslices and Aggregation

A human analyst may extend the analysis tool with bitslices and aggregation algorithms specific to the chip being analyzed. We used this technique to identify the VGA frame buffer structure in the BigSoC design.

6.2.2 Manual Analysis of Candidate Modules

Besides fully identified modules, the tool can also be made to output “candidate” modules generated by common signal aggregation of “unknown” bitslices (§2.2). A human analyst can look at the generated modules and try to understand their functionality, for example, by simulating with random inputs. Analyzing these modules is easier than analyzing the entire chip because: (i) the modules only have a few tens or hundreds of gates and (ii) the modules have regularity and structure unlike the full netlist.

6.2.3 Manual Analysis of Uncovered Gates

We can derive useful information about the functionality of unidentified gates using the output of the tool. Two of the counters identified in the router are actually head and tail pointers which index into a FIFO. Knowing these are counters helped understand the functionality of the indexing structure. Another case is of structures that do not have a clean interconnection pattern but have replicated bitslices that can be detected using cut-based Boolean matching.18

6.2.4 Simulation-Based Analyses

The techniques in this paper focuses entirely on “static” analysis of the netlist. Simulation of the netlist with carefully constructed test vectors is a form of dynamic analysis that can provide valuable information. For instance, one conceivable way of detecting an FFT co-processor is to construct a test program executing FFTs in a loop, simulating its execution and observe where the (known) operands and results of the transform show up. We are working on such algorithms.

6.2.5 Discussion And Comparison with Previous Work

This paper introduced a portfolio of algorithms for reverse engineering gate-level netlists. It builds on our previous work [26, 13] in this area. The work in [26] introduced bitslice matching and aggregation and provided a brief overview of the algorithms for counter, shift register and RAM detection. This work adds new algorithms based on analyzing nodes with common support (§2.5), the multibit register analysis (§3.4) and the ILP formulation to resolve overlapping output modules (§4). This paper has also integrated the functional word propagation algorithm (§2.3) and QBF-based module matching algorithm (§2.4) from [13]. These algorithms have proven to be more effective than the structural word propagation and BDD-based module matching algorithms presented in [26]. This paper also expanded on the descriptions of the algorithms for detecting counters (§3.1), shift registers (§3.2) and RAMs (§3.3). The evaluation of our algorithms (§5) in this paper is much more detailed. In particular, we believe that the detailed analysis of the BigSoC design (§5.3) and the partitioning algorithms used in making the analysis of BigSoC tractable significant contributions of this paper. The trojan detection experiments from §5.4 which demonstrate the feasibility of trojan detection

18. This happens for less-than/greater-than comparison circuits.
aided by algorithmic reverse engineering are also an important novel contribution of this paper.

7 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 which: (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 45% to 93% of the gates in a netlist may be automatically inferred using our algorithms. We also demonstrated that our algorithms achieve 68% coverage on a large highly-optimized SoC consisting of over 375,000 gates. We also demonstrated that these algorithms are very effective in aiding a human analyst detect hardware trojans in an unstructured netlist.

References