Reverse Engineering Circuits Using Behavioral Pattern Mining

Wenchao Li
UC Berkeley
wenchaol@eecs.berkeley.edu

Zach Wasson
UC Berkeley
zwasson@eecs.berkeley.edu

Sanjit A. Seshia
UC Berkeley
sseshia@eecs.berkeley.edu

ABSTRACT

Systems are increasingly being constructed from off-the-shelf components acquired through a globally distributed, untrusted supply chain. The lack of trust in these components necessitates additional validation of the components before use. Additionally, hardware trojans are becoming a pressing concern. In this paper, we present a novel formalism and method to systematically derive the high-level function of an unknown circuit component given its gate-level netlist. We define the high-level description of a circuit as an interconnection of instantiations of abstract library components characterized using logical specifications. The proposed approach is based on mining interesting behavioral patterns from the simulation traces of a gate-level netlist, and representing them as a pattern graph. A similar pattern graph is also generated for library components. Our method first computes input-output signal correspondences via subgraph isomorphism on the pattern graphs. The general function of the unknown circuit is then determined by finding the closest match in the component library, by model checking the unknown circuit against each logical specification. We demonstrate the effectiveness of our approach on publicly-available circuits.

1. INTRODUCTION

Systems are increasingly being constructed from off-the-shelf components acquired through a globally-distributed, untrusted supply chain. The lack of trust in these components implies that they must be validated before use. Additionally, hardware trojans are becoming a pressing concern. One approach to address the problem, in both cases, is to reverse-engineer the high-level functionality of the circuit. In order to do this, there is a need for both a formalization of the problem – reverse engineering a high-level description (HLD), as well as algorithmic techniques to derive a HLD from the original circuit.

One of the challenges is simply to come up with a suitable formal problem definition. We restrict ourselves, in this paper, to digital circuits. Several steps must be completed in order to define the reverse engineering problem and solve it. First, one must define what "high-level" means. Intuitively, a high-level characterization of a circuit can be one that identifies the large functional blocks comprising the circuit, along with their interconnection. The space of possible functional blocks can be defined using a library of high-level components, where a component is either a commonly-used hardware design pattern or a custom finite-state machine (FSM) block. Any circuit composed of such components can be described in various ways, e.g., as a high-level netlist, or in a suitable hardware description/programming language. Second, we need a method of identifying candidate blocks in the unknown circuit to match against the high-level component library. Third, we must compute a correspondence between input and output signals of the unknown block and those of high-level components. Fourth, given such a correspondence, we must formally define when an unknown block is said to match a given high-level component. Fifth, given this definition of component matching, we must verify whether the unknown block indeed matches the high-level component. Finally, such a match must be computed for all candidate blocks identified from the unknown circuit, and the results combined to generate the final high-level description.

In this paper, we present a formalization of the problem of reverse engineering a high-level description (henceforth referred to as REHLD) of a digital circuit, along with an initial approach to solve it. The context we address is that of a system integrator needing to reverse-engineer the functionality of a component she did not design. Our problem formalization is based on two main insights. First, what constitutes a high-level component depends on the view of the system integrator, but many standard hardware design patterns can be useful in the reverse engineering process. For example, components such as arbiters, FIFOs, adders, and register files are common hardware design patterns that appear in many designs. Similarly, standard protocols are often used in buses and interconnection networks and the control logic for these structures yield high-level FSM components. Second, the high-level components must be specified in an abstract, behavioral way, using mathematical, logical specifications. One can have many FIFO buffer implementations, but all of them share the basic property of first-in, first-out data storage and transmission. Thus, an abstract component is really a specification of a family of concrete components, all of which share a common set of properties.

We combine the formalism with an initial approach to systematically derive the high-level function of an unknown digital circuit, given its gate-level netlist. The proposed approach is based on mining interesting behavioral patterns from the simulation traces of a gate-level netlist, and represents them as a pattern graph. A similar pattern graph is also generated for library components. Our method first computes input-output signal correspondences via subgraph isomorphism on the pattern graphs. The general function of the unknown circuit is then determined by finding the closest match in the component library, by model checking the unknown circuit against each logical specification.

To summarize, the contributions of this paper include:

- A formal framework for the reverse engineering problem, based on the notion of matching against *abstract library components* (Section 2), and
- An approach for matching an unknown sub-circuit against an abstract component library based on *mining behavioral patterns* from simulation or execution traces, followed by model checking (Section 3).

We demonstrate our approach on publicly-available benchmarks, including implementations of the serial peripheral interface (SPI) bus standard (Section 4).

2. PROBLEM FORMULATION

We begin with some basic definitions and terminology in Sec. 2.1, followed by the formal problem statement in Sec. 2.2.

2.1 Definitions and Terminology

2.1.1 Circuit and Traces

A *bit-level netlist or circuit* $\mathcal{C}$ is a tuple ($\mathcal{I}$, $\mathcal{O}$, $\mathcal{V}_S$, $\mathcal{V}_C$, $Init$, $\mathcal{A}$) where

978-1-4673-2340-6/12/$31.00

- $\mathcal{I}$ is a finite set of input signals;
- $\mathcal{O}$ is a finite set of output signals;
- $\mathcal{V}_\mathcal{S}$ is a finite set of intermediate sequential (state-holding) signals;
- $\mathcal{V}_\mathcal{C}$ is a finite set of intermediate combinational (stateless) signals;
- *Init* is a set of initial states, i.e., initial valuations to elements of $\mathcal{V}_\mathcal{S}$, and
- $\mathcal{A}$ is a finite set of assignments to outputs and to sequential and combinational intermediate signals. An assignment is an expression that defines how a signal is computed and updated.

All signals are Boolean. A combinational assignment is denoted by $s \leftarrow e$, where s is a signal and e is a Boolean expression. Similarly, a sequential assignment is denoted by $s := e$. Input and output signals are assumed to be combinational, without loss of generality.
An *input-output trace* (or simply, a *trace*) of a circuit is a sequence of valuations $\vec{v}_0, \vec{v}_1, \vec{v}_2, \vec{v}_3, \ldots$ to input and output signals; i.e., each $\vec{v}_i$ is a vector of 0-1 values to variables in $\mathcal{I} \cup \mathcal{O}$, and the subscript i denotes the cycle at which the valuation is recorded. For simplicity, we restrict ourselves to valuations of signals at the rising edge of the clock; the results of the paper extend to other conventions as well. A *finite trace* of a circuit is a finite sequence of input-output valuations $\vec{v}_0, \vec{v}_1, \vec{v}_2, \vec{v}_3, \ldots, \vec{v}_k$.
An *event* e is a tuple $\langle \vec{s}, \vec{v}, t \rangle$, where $\vec{s}$ is a set of signals and $\vec{v}$ is the corresponding valuations at cycle t. We denote the valuation of a Boolean signal s at cycle t as $v_{s,t}$.
Note that we do not define events as assignments of signals across cycles, but this can be addressed by introducing suitable user-defined events.
A *delta event*, denoted Δe, is an event such that at least one of its constituent signals changes value from the previous valuation, i.e. $\Delta e := \langle \vec{s}, \vec{v}, t \rangle$ such that $\exists\ s \in \vec{s}, v_{s,t} \neq v_{s,t-1}$. For a Boolean signal s, we use $s_{0\to1}$ to denote the delta event of s transitioning from 0 to 1 and similarly $s_{1\to0}$ to denote the delta event of s transitioning from 1 to 0.

2.1.2 Formal Specification

A *formal specification* of a sequential circuit $\mathcal{C}$ is a set $\mathcal{S}$ of input-output traces of that circuit. Intuitively, every trace in $\mathcal{S}$ is an allowed behavior of $\mathcal{C}$ and every trace outside $\mathcal{S}$ is disallowed.
There are broadly two ways to write a formal specification for a digital circuit. The automata-theoretic approach is to describe $\mathcal{S}$ as a finite-state machine over infinite input sequences [22]. The other approach is to write a logical formula (or set of formulas) characterizing the input-output behavior of the circuit. The latter approach has gained favor in the EDA community over the years, especially in the form of assertion languages that allow one to specify temporal properties of a system, and which are usually slight extensions of *linear temporal logic* (LTL) [18]. We also follow this logic-based approach in our work.
A brief overview of LTL is provided below.
We can express that "every request must be eventually followed by a grant" in LTL as "$\mathbf{G}$ (request $\Rightarrow \mathbf{F}$ grant)", where the operator $\mathbf{G}$ specifies that globally at every point in time a certain property holds, and $\mathbf{F}$ specifies that a property holds either currently or at some point in the future.
In this paper, we mainly write specifications using a combination of LTL and regular expressions, employing the following four patterns:

1. **Alternating (A)** An alternating pattern between two delta events Δa and Δb is true when each occurrence of Δa alternates with an occurrence of Δb. Note that this does not mean Δb follows Δa immediately in the next cycle. This pattern can be described by the regular expression $(\Delta a\ \Delta b)^*$. Figure 1 shows the corresponding finite automaton (self-transitions are such that the automaton is deterministic).
2. **Next (X)** The next pattern corresponds to the LTL formula "$\mathbf{G}\ (\Delta a \Rightarrow \mathbf{X}\ \Delta b)$." One can easily generalize this pattern to fixed-delay pairs.
3. **Until (U)** The until pattern can be used to describe behaviors such as "the request line stays high until a response is received." Figure 2 shows a trace where this pattern is satisfied. Formally, the LTL formula is "$\mathbf{G}\ (a_{0\to1} \Rightarrow \mathbf{X}\ (a\ \mathbf{U}\ b_{0\to1}))$."

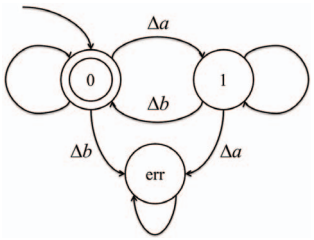


Figure 1: Finite Automaton for the Alternating Pattern

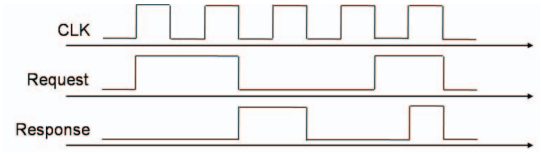


Figure 2: Request stays high until a response is received.

4. **Eventual (F)** The eventual pattern can be described by the LTL formula "$\mathbf{G}\ (\Delta a \Rightarrow \mathbf{X}\ \mathbf{F}\ \Delta b)$."

It is important to note that, even though we mainly focus on the above temporal logic patterns, the problem formulation described in this paper applies to all types of formal specifications.

2.1.3 High-Level Description

Informally, a high-level description of a circuit is a composition of high-level *abstract components*.
An *abstract component* α is a triple $(\mathcal{I}, \mathcal{O}, \mathcal{S})$, where $\mathcal{I}$ and $\mathcal{O}$ are sets of input and output signals, respectively, and $\mathcal{S}$ is a formal specification defining allowed input-output behavior of the circuit.
An *instance* of an abstract component α is any circuit that satisfies the specification $\mathcal{S}$ of α.
We illustrate the notion of an abstract component using an example.

EXAMPLE 1. *Consider an arbiter servicing two (input) request lines r_0 and r_1 with two grant lines g_0, g_1 as outputs. The abstract arbiter component would comprise the following LTL properties:*

$$\mathbf{G}\ \neg(g_1 \wedge g_2)$$
$$[\mathbf{G}\ \mathbf{F}\ \neg(r_0 \wedge r_1)] \Rightarrow \mathbf{G}\ (r_0 \Rightarrow \mathbf{F}\ g_0)$$
$$[\mathbf{G}\ \mathbf{F}\ \neg(r_0 \wedge r_1)] \Rightarrow \mathbf{G}\ (r_1 \Rightarrow \mathbf{F}\ g_1)$$

The first property states that both requests cannot be granted at the same cycle. The last two properties state that a request must eventually be granted, provided there are infinitely many cycles where no competing requests are present — the latter assumption is the property $\mathbf{G}\ \mathbf{F}\ \neg(r_0 \wedge r_1)$. □

A *library* of abstract components (*component library*, for short) $\mathcal{L}$ is a set $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ of abstract components. We will assume that each abstract component is also accompanied by at least one concrete instance; this is reasonable, as the abstract component library is typically constructed from observations of commonly occurring components in hardware designs.

EXAMPLE 2. *Examples of abstract components include common hardware design patterns and modules such as an arbiter,*

FIFO buffer, adder, multiplier, content-addressable memory (CAM), and crossbar. Abstract descriptions of finite-state machines are relevant for circuits that implement protocols; e.g., transmitter or receiver modules implementing the Ethernet or I^2C protocols. □

A *high-level description* (HLD) of a circuit $\mathcal{C}$ with inputs $\mathcal{I}$ and output $\mathcal{O}$ is a tuple $(\mathcal{I}, \mathcal{O}, \Gamma)$ where Γ is a set of instances of abstract components drawn from $\mathcal{L}$ such that the *synchronous composition* of these instances is (sequentially) *equivalent* to the original netlist $\mathcal{C}$.

EXAMPLE 3. *Consider the HLD of a chip multiprocessor (CMP) router as shown in Figure 3. It is a composition of four high-level modules. The* input controller *comprises a set of FIFOs buffering incoming flits and interacting with the* arbiter. *When the arbiter grants access to a particular output port, a signal is sent to the input controller to release the flits from the buffers, and at the same time, an allocation signal is sent to the* encoder *which in turn configures the* crossbar *to route the flits to the appropriate output port.*

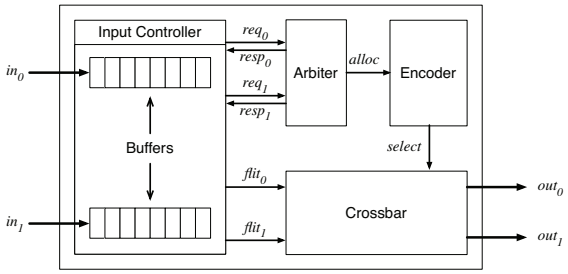


Figure 3: CMP Router comprising four high-level modules

2.2 Problem Definition

We are now ready to formally define the problem of reverse engineering a high-level description (REHLD) of a circuit.

DEFINITION 1. *Given*

- *an* unknown *bit-level circuit $\mathcal{C}$ = ($\mathcal{I}$, $\mathcal{O}$, $\mathcal{V}_S$, $\mathcal{V}_C$, Init, $\mathcal{A}$), and*
- *a library $\mathcal{L} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ of abstract components,*

the REHLD problem for $\mathcal{C}$ is to derive a high-level description $\mathcal{C}' = (\mathcal{I}, \mathcal{O}, \Gamma)$ where $\mathcal{C}'$ is equivalent to $\mathcal{C}$ and Γ is a composition of instances of abstract components from $\mathcal{L}$.

As noted in Section 1, solving the REHLD problem involves multiple steps. We rephrase these steps using the notation introduced in this section. There are three main steps:

1. *Library definition:* Constructing an abstract component library $\mathcal{L}$;
2. *Functional block identification:* Dividing the given unknown circuit $\mathcal{C}$ into a set of functional blocks (sub-circuits) $b_1, b_2, \ldots, b_k$, where each b_i is considered a candidate for matching against the component library $\mathcal{L}$;
3. *Matching against component library:* Given a candidate functional block (sub-circuit) b_i and an abstract component library $\mathcal{L} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, determine whether there exists an abstract component α_j such that:
   (a) *Input-output correspondence:* Each input (respectively, output) signal of α_j that appears in the formal specification of α_j is mapped 1-1 to some input (respectively, output) signal of b_i.
   (b) *Verification:* b_i is an instance of α_j; i.e., b_i satisfies the specification $\mathcal{S}_j$ associated with α_j.

In the rest of this paper, we address some of the key steps in the above process. Step 1, library definition, depends on the requirements of the end user of the reverse engineering procedure. We use a representative sample of communication protocols in our experiments. In general, a burden is placed on the end user to create an initial component library. While this is a limitation for any library-based approach, common hardware design patterns [8] and standardized interfaces are ideal starting components for the library. We do not address Step 2 (functional block identification) in this paper, leaving it to future work. The major component addressed by the remainder of the paper is Step 3. Specifically, for 3(a), we show how temporal patterns mined from simulation or execution traces of the circuits can be used to determine input-output correspondence. For 3(b), we use *model checking* [7] to determine whether a given bit-level circuit satisfies the formal specification associated with an abstract component.

3. APPROACH

In this section, we give a detailed description of our approach based on behavioral pattern mining and model checking.

3.1 Input-Output Correspondence

Given an unknown sub-circuit $\mathcal{C}$ and an abstract library component α, we must first compute the correspondence between their input and output signals, if one exists. A brute-force approach will have to try all possible permutations of the signals. In addition, the numbers of inputs and outputs of the two circuits may not be identical. We use a heuristic procedure for this step. First, a concrete instance of α (e.g. a reference circuit that satisfies that formal specification of α), denoted $\mathcal{C}'$ is obtained. Then, given the two circuits $\mathcal{C}$ with interface signals $\mathcal{V}_\mathcal{I} = \mathcal{I} \cup \mathcal{O}$ and $\mathcal{C}'$ with interface signals $\mathcal{V}_\mathcal{I}' = \mathcal{I}' \cup \mathcal{O}'$, our method tries to find the corresponding signals between $\mathcal{V}_\mathcal{I}$ and $\mathcal{V}_\mathcal{I}'$. This problem is formally defined as follows.

DEFINITION 2. *Given two sets of signals A and B, the signal correspondence problem is to find a bijective mapping $\sigma : \bar{A} \rightarrow \bar{B}$, where $\bar{A} \subseteq A$ and $\bar{B} \subseteq B$. We say the mapping is* maximum *if there does not exist another mapping $\sigma' : \tilde{A} \rightarrow \tilde{B}$ such that $\bar{A} \subset \tilde{A} \subseteq A$ or $\bar{B} \subset \tilde{B} \subseteq B$.*

Our approach to solving the signal correspondence problem is based on mining patterns from a set of input-output traces of the two circuits. Other approaches, e.g., based on the structure of the circuits, are also possible, and will be left to future work.

The key insight in our approach is that two signals are *similar* if they exhibit similar behaviors in relation to other signals. In this paper, we measure the similarity of two signals by checking if they satisfy some particular patterns (in relation to other signals) in the traces. However, our framework is general – one can use other definitions of similarity such as statistical measures.

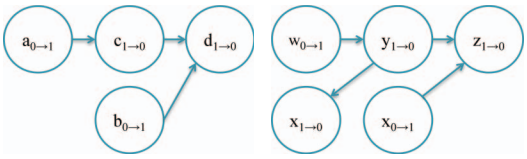
In a nutshell, our method uses a two-step combination of *pattern mining* and *graph matching*. The pattern mining step infers *likely* temporal properties of a circuit from input-output traces of that circuit. It is important to note here that for the unknown component, we can only observe the trace induced by a test bench at the chip-level. This means that it is not possible to control the simulation as in the case of the known component. Consequently, even if the unknown circuit is identical to the library component, the behaviors of the two traces can be very different. In this work, we use pattern mining as a way to concisely capture key features of a trace. The mined properties are represented in terms of a *pattern graph*. Given circuits $\mathcal{C}$ and $\mathcal{C}'$, we compute pattern graphs for each of them. Then, a graph matching procedure is used to compute the *maximum common subgraph* between these two graphs, which yields the desired maximum bijection. We elaborate below.

3.1.1 Pattern Mining

Given a set of input-output traces and a pattern template as described in Section 2.1.2, it first generates a pattern graph $G = \langle V, E \rangle$ where $E \subseteq V \times V$. A vertex $v \in V$ represents a delta event of some signal in $\mathcal{I} \cup \mathcal{O}$. For example, a vertex labeled with $\Delta a_{0\to1}$ represents the event of signal a transitioning from 0 to 1. There is a directed edge $e = (u, v) \in E$ if and only if the pattern involving the delta events represented by u and v is satisfied by all traces. For example, given the pattern template "$\mathbf{G}\,(\Delta a \Rightarrow \mathbf{X}\,(a\,\mathbf{U}\,\Delta b))$," there is an edge from a vertex labeled with $\Delta a_{0\to1}$ to a vertex labeled with $\Delta b_{0\to1}$ if and only if the pattern is satisfied by the set of traces.

As an example, let α be the arbiter described in Section 2.1.3. Suppose $\mathcal{C}$ uses a round-robin priority scheme and $\mathcal{C}'$ uses a fixed priority scheme for arbitration.

Given the input-output traces of $\mathcal{C}$ and $\mathcal{C}'$ and the **Until** pattern, suppose that Figure 4 shows the pattern graphs generated for $\mathcal{C}$ and $\mathcal{C}'$.[1]



(a) Round-robin priority (b) Fixed priority

Figure 4: Arbiter Pattern Graphs

Table 1 shows how the named signals a-d of the round-robin priority arbiter and w-z of the fixed-priority arbiter correspond to request/grant signals r_0,r_1,g_0, and g_1 as described in Sec. 2.1.3. (Our task is to compute this correspondence, but we provide it here so that the reader can follow this example.) Thus, for example, the edges from $a_{0\to1}$ to $c_{0\to1}$ and from $w_{0\to1}$ to $y_{0\to1}$ are instances of the property $\mathbf{G}\,[r_{00\to1} \Rightarrow \mathbf{X}\,(r_0\,\mathbf{U}\,g_{0\,0\to1})]$ (request stays high until the corresponding grant is asserted).

Table 1: Signal Names in Arbiter Versions

Signals	Round-robin	Fixed
r_0	a	w
r_1	b	x
g_0	c	y
g_1	d	z

3.1.2 Graph Matching

A graph $G' = \langle V', E' \rangle$ is an *induced subgraph* of $G = \langle V, E \rangle$ if $V' \subseteq V$ and $E' = E \cap (V' \times V')$. G is said to be *isomorphic* to G' if there exists a bijective function $f : V \to V'$ such that $\forall\,(u, v) \in V \times V, (u, v) \in E \iff (f(u), f(v)) \in E'$.

Given the two pattern graphs G and G' corresponding to $\mathcal{C}$ and $\mathcal{C}'$ respectively, a *common subgraph* is a graph which is *isomorphic* to induced subgraphs of G and G'. We wish to find a *maximum common subgraph* (MCS) between the two graphs, i.e., a common subgraph between G and G' that has the maximum number of vertices.

The key observation here is that the bijective function f that defines the MCS is exactly the signal correspondence mapping σ that we are looking for. In fact, since the vertices represent delta events, our approach can identify corresponding signals even if they are implemented with opposite polarities in the two circuits.

[1]In the case where a pattern graphs is composed of multiple disconnected subgraphs, we use the biggest subgraph as the pattern graph.

Consider our arbiter example. Figure 5 shows a maximum common subgraph of the pattern graphs given in Figure 4. Hence, all

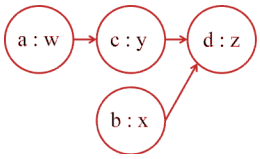


Figure 5: MCS in the two arbiter pattern graphs

the request and response signals are mapped correctly using MCS approach.

The MCS problem is known to be NP-hard [10]. Most complete approaches are based on reformulating the problem into a *maximum clique* problem in a *compatibility graph* between G and G' [9]. The compatibility graph is a product graph of G and G' such that a vertex in the product graph is a pair of vertices (i, k) where $i \in V$ and $k \in V'$. There is an edge from (i, k) to (j, l) ($i \neq j$ and $k \neq l$) if and only if one of the following conditions hold:

- $(i, j) \in E$ and $(k, l) \in E'$;
- $(i, j) \notin E$ and $(k, l) \notin E'$.

In a directed graph G, we say two vertices u and v are *connected* if both $(u, v) \in E$ and $(v, u) \in E$. A *clique* is then a subset of vertices such that every pair of vertices in this set are connected. A *maximum clique* is a clique with the most number of vertices.

We omit details of how the maximum clique problem is solved but refer the readers to [6]. The technique can scale to graphs with hundreds of vertices. Solving the maximum clique problem correctly generates the signal correspondence as depicted in Table 1.

3.2 Matching by Verification

If all inputs and outputs of the abstract component α appearing in its specification $\mathcal{S}$ are mapped onto some inputs and outputs of the unknown circuit $\mathcal{C}$, then we proceed to the next step, where we verify whether $\mathcal{C}$ satisfies $\mathcal{S}$. In our approach, we perform this step using model checking [7], as $\mathcal{S}$ is a set of LTL properties. (If $\mathcal{S}$ were a reference circuit, it is possible to replace the model checker with a sequential equivalence checker.) If $\mathcal{C}$ satisfies $\mathcal{S}$, then we terminate reporting that $\mathcal{C}$ matches the component α. Otherwise, we report that it does not match (and move to trying to match $\mathcal{C}$ to a different abstract component).

However, if some inputs/outputs of α appearing in $\mathcal{S}$ are *not* matched to inputs/outputs of $\mathcal{C}$, then we stop and declare that there is no match between $\mathcal{C}$ and α. Note that this approach is *conservative*: it is possible for $\mathcal{C}$ to be an instance of α but not pass this phase, since our input-output signal correspondence algorithm is heuristic. However, it is important to note that our matching approach is *sound* due to the use of formal verification.

While formal verification may not scale to the full circuit, we envisage applying our procedure mainly to blocks with hundreds to thousands of flip-flops, which is within the capacity of state-of-the-art model checkers today. We demonstrate our approach on benchmarks from OpenCores, as discussed in the following section.

Lastly, it should be noted that while it may be difficult to write specifications that account for all possible behaviors of each library component, it is generally possible to distinguish one design block from another using only a few logical specifications (e.g. distinguish an arbiter from an adder). We believe the automation that we provide in the proposed approach can benefit reverse-engineering greatly, as it is still mostly a manual process today.

4. RESULTS

We used the following three circuits obtained from OpenCores [2] in our experiments.

- WISHBONE-compatible SPI (WB-SPI) [5]
- WISHBONE-compatible SimpleSPI (WB-SimpleSPI) [3]
- WISHBONE-compatible I^{2}C (WB-I^{2}C) [1]

The serial peripheral interface (SPI) is a full duplex, serial communication link. Devices operate in either master or slave mode. Typically there is a single master device and one or more slave devices. SPI specifies four logic signals: SCLK (serial clock), MOSI (master output/slave input), MISO (master input, slave output), and SS (slave select). The SS signal is only necessary if more than one slave device is connected to the master. To initiate a data transfer, the SS pin for the desired slave is first pulled low. Then data are clocked from the master to the slave via the MOSI port and data are clocked from the slave to the master via the MISO port. When the transfer is complete, the SS pin is pulled high. SimpleSPI is a simplified version which supports only one master and one slave.

The inter-integrated circuit or I^{2}C is a serial two-wire communication bus. Devices operate in either master or slave mode, similar to SPI; however, there can be multiple masters on an I^{2}C bus. The bus is made up of two logic signals: SCL (clock) and SDA (data). A typical data transfer starts with a master sending a START bit along with a 7-bit address for the slave it wishes to communicate with and a bit indicating a read or write operation. The slave responds with an ACK bit and proceeds to operate in either read mode or write mode, depending on the master's request. Once the transmission is over, the master sends a STOP bit.

WISHBONE is a communication interface for IP cores that enhances design reuse by enforcing compatibility between cores. Furthermore, WISHBONE is open source, which makes it easy for engineers to share hardware designs. As such, many projects on OpenCores, a website dedicated to open source hardware designs, include a WISHBONE interface. The protocol supports handshaking, single read/write cycles, block read/write cycles, and read-modify-write cycles. All three circuits are supposed to be WISHBONE compatible.

We consider the scenario where the WISHBONE protocol [4] has been pre-characterized as a library component and the WB-SPI circuit is given as a concrete implementation of the WISHBONE protocol. We treat the WB-SimpleSPI and WB-I^{2}C as unknown candidate functional blocks. The goal is to determine whether an unknown circuit also implements the WISHBONE interface.

Signal Correspondence:

We followed the approach in [13] for generating the pattern graph by mining the **Until** pattern on the simulation traces of each circuit. The simulation traces were produced by the test benches provided on OpenCores. We further assume the clock, reset (wb_rst_i) and data-path signals at the interface are already identified (these are easy to identify by structural methods). Next, a compatibility graph was generated for the pattern graph of the unknown circuit and WB-SPI.[2] The time taken to generate each pattern graph was under a second. We used the program `Cliquer` [17] which implements an exact branch-and-bound algorithm developed by Patric Östergård for finding the maximum clique in a graph. The time taken to find a maximum clique in the compatibility graph was under a second in all instances.

Table 2 shows the signal map derived between SPI and SimpleSPI by using the first MCS produced. All the WB-related signals were mapped correctly. In addition, two SPI-related signals were also mapped correctly. The signal names used here were taken from the respective RTL files and only serve as an illustration. In an actual reverse engineering exercise, the signal names of the unknown circuit will be arbitrary. These results were obtained despite the fact that the traces were generated by two test benches with very different behaviors.

[2]Observe that the pattern graph only needs to be generated once for the library component.

Table 2: Signal Mapping between WB-SPI and WB-SimpleSPI

WB-SPI	WB-SimpleSPI
$miso\_pad\_i$	$miso\_i$
wb_ack_o	ack_o
$sclk\_pad\_o$	sck_o
wb_we_i	we_i
wb_cyc_i	cyc_i
wb_stb_i	stb_i

Table 3 shows the signal map derived between SPI and I^{2}C by also using the first MCS produced.

Table 3: Signal Mapping between WB-SPI and WB-I^{2}C

WB-SPI	WB-I^{2}C
wb_we_i	wb_we_i
wb_cyc_i	wb_cyc_i
wb_ack_o	wb_ack_o

All the WB-related signal were again matched correctly. However, the wb_stb_i signal was not matched. It was identified by iterating the approach with the **Alternating** pattern, given the current matches. This suggests an incremental approach to our framework but this will be left to future work. On the other hand, there were also four other incorrectly matched signals. This is because other than being both WISHBONE compatible, the two circuits were actually implementing different functions. The next step which checks these signals against their logical specifications will eliminate the incorrect matches.

Matching by Verification:

We focused on specifying the slave interface of the WISHBONE protocol, although we used properties of the master interface as assumptions.

At a minimum, a slave interface requires the following signals: wb_ack_o, wb_clk_i, wb_cyc_i, wb_stb_i, and wb_rst_i. Since the WISHBONE interface does not explicitly require the data lines, wb_dat_o and wb_dat_i, the only properties we could specify were about the reset operation and the handshaking protocol.

The properties for the reset operation and handshaking protocol are as follows:

- $\mathbf{G}\ (wb\_rst\_i \Rightarrow \mathbf{X}\ \neg wb\_ack\_o)$
- $\mathbf{G}\ \neg(wb\_ack\_o \wedge \mathbf{X}\ wb\_ack\_o)$
- $\mathbf{G}\ ((wb\_cyc\_i \wedge wb\_stb\_i) \Rightarrow \mathbf{F}\ wb\_ack\_o)$

The assumptions we made on the master interface, part of the specification, are as follows:

- $\mathbf{G}\ (wb\_rst\_i \Rightarrow (\neg wb\_stb\_i \wedge \neg wb\_cyc\_i))$
- $\mathbf{G}\ (wb\_stb\_i \Rightarrow wb\_cyc\_i)$

These LTL properties were manually translated from the WISHBONE documentation Rev. B.4 [4].

Taking the netlist descriptions of WB-SimpleSPI and WB-I^{2}C, we translated them to the SMV format and verified the properties described above using the Cadence SMV model checker [14]. For both circuits, all the properties passed. This confirmed that both indeed implemented the WISHBONE interface. The verification times were under 0.1 second for either benchmark.

5. RELATED WORK

Digital system designers usually proceed from a high-level description to a gate-level netlist, and then to a physical layout and mask; it is rare to proceed in the opposite direction. However, as noted in Sec. 1, the study of reverse engineering of digital circuits has been gaining importance in recent years. We review here the most closely related work.

Hansen et al. [11] present a study of reverse engineering the well-known ISCAS-85 combinational circuits. They present several strategies, mostly manual, to reverse engineer circuit functionality from a gate-level schematic. Some of these include looking for common library components, repeated structures, computing truth tables of small blocks, and identifying bus structures and control signals. However, they do not formally characterize the problem. To the best of our knowledge, ours is one of the first formal definitions of the reverse engineering problem. The component library in our work is at a much higher level of abstraction than that suggested by Hansen et al. Moreover, our component matching is automated and operates on sequential circuits.

Torrance and James [23] describe the practice of reverse engineering semiconductor-based products. Their approach includes product tear-downs (stripping packaging and disassembling the unit), "system-level analysis" (identifying components on a board and performing functional analysis through probing), process analysis, and circuit extraction (deriving a schematic from a stripped IC). Our work is complementary to this effort. Once a gate-level schematic is derived, our techniques can be applied to match modules within the schematic to an abstract component library.

Our work does not address reverse engineering of arbitrary finite-state machine functionality. While any sequential circuit can be trivially viewed as a monolithic FSM, the challenge is to be able to decompose that FSM into a set of smaller FSMs, each of which performs a distinguishable function, thus making the resulting high-level description easier for a human to understand. The recent work by Shi et al. [20] is a step in this direction.

A key component of our work is to find the input-output signal correspondence between an abstract component and a block in the unknown circuit. There is not very much prior work in this area. Mohnke and Malik [15] present a BDD-based approach for comparing two combinational circuits whose input correspondence is not known apriori. The authors also extended their idea to finding latch correspondence for sequential circuits, by considering the combinational circuit computing the next-state function [16]; this does not address our problem, though, as sequential equivalence is required between the two circuits. Our approach, based on mined temporal properties and graph matching, is novel.

Our technique addresses the REHLD problem for a system integrator who has *not* designed the circuit being reverse-engineered, but instead needs to verify its functionality prior to integration. We do not address the problem of untrusted manufacturing and IC piracy, where the designer is trusted, which can be tackled by techniques such as EPIC [19]. Our technique is complementary to other recent work on malicious trojan circuit detection (e.g., [12, 21]). We do not seek to find trojans, instead focusing on detecting if a sub-circuit exhibits correct behavior which is captured by a set of logical specifications; if our approach deems a sub-circuit to match an abstract component, it is guaranteed to do so due to the use of formal verification. In addition, if the sub-circuit violates some security-related specification, our approach will report that as well.

6. CONCLUSION

We presented a new formal definition of the problem of reverse engineering a high-level description of an unknown digital circuit. A solution strategy was sketched out and a new technique proposed for the key step of matching a block in the unknown circuit to an abstract component library. Our technique is based on a combination of pattern mining from input-output traces and model checking. Experimental results demonstrate the promise of this approach.

There are several directions for future work. Structural techniques can complement our behavioral approach to input-output signal correspondence. New methods must be devised to find candidate functional blocks in the overall circuit to match against the component library. Additionally, to complete the reverse engineering process, one would need to integrate the block identification and matching procedures into an iterative loop, that finds the best "covering" of the unknown circuit with abstract high-level components.

Acknowledgements. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under the IRIS program, and by the Hellman Family Faculty Fund.

7. REFERENCES

[1] I2c controller core. http://opencores.com/project,i2c.
[2] Opencores. http://opencores.com/.
[3] Simple spi core. http://opencores.com/project,simple_spi.
[4] Soc interconnection: Wishbone. http://opencores.org/opencores,wishbone.
[5] Spi core. http://opencores.com/project,spi.
[6] E. Balas and C. S. Yu. Finding a maximum clique in an arbitrary graph. *SIAM J. Comput.*, 15:1054–1068, November 1986.
[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
[8] A. DeHon, J. Adams, M. DeLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, and M. Wrighton. Design patterns for reconfigurable computing. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 13 – 23, april 2004.
[9] P. J. Durand, R. Pasari, J. W. Baker, and C. Tsai. An efficient algorithm for similarity analysis of molecules. *Internet Journal of Chemistry*, 1999.
[10] M. R. Garey and D. S. Johnson. *Computer and intractability*. Freeman, 1979.
[11] M. C. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers*, 16(3):72–80, 1999.
[12] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *IEEE Symposium on Security and Privacy*, pages 159–172, 2010.
[13] W. Li, A. Forin, and S. A. Seshia. Scalable specification mining for verification and diagnosis. In *Design Automation Conference*, pages 755–760, 2010.
[14] K. McMillan. The cadence smv model checker. http://www.kenmcmil.com/smv.html.
[15] J. Mohnke and S. Malik. Permutation and phase independent boolean comparison. In *European Conference on Design Automation*, Feb. 1993.
[16] J. Mohnke, P. Molitor, and S. Malik. Establishing latch correspondence for sequential circuits using distinguishing signatures. In *MidWest Symposium on Circuits and Systems*, pages 472–476, 1997.
[17] S. Niskanen and P. Östergård. Cliquer - routines for clique searching. http://users.tkk.fi/pat/cliquer.html.
[18] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.
[19] J. A. Roy, F. Koushanfar, and I. L. Markov. EPIC: Ending piracy of integrated circuits. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1069–1074, 2008.
[20] Y. Shi, C. W. Ting, B.-H. Gwee, and Y. Ren. A highly efficient method for extracting FSMs from flattened gate-level netlist. In *IEEE International Symposium on Circuits and Systems (ISCAS 2010)*, pages 2610–2613, 2010.
[21] C. Sturton, M. Hicks, D. Wagner, and S. T. King. Defeating UCI: Building stealthy and malicious hardware. In *IEEE Symposium on Security and Privacy*, pages 64–77, 2011.
[22] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, pages 133–164. Elsevier, 1990.
[23] R. Torrance and D. James. The state-of-the-art in IC reverse engineering. In *11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 5747 of *Lecture Notes in Computer Science*, pages 363–381, 2009.