Simulation Graphs for Reverse Engineering

Mathias Soeken[1,2,4] Baruch Sterin[3] Rolf Drechsler[2,4] Robert Brayton[3]
[1] Faculty of Engineering, University of Freiburg, Freiburg, Germany
[2] Faculty of Mathematics and Computer Science, University of Bremen, Germany
[3] Electrical Engineering and Computer Sciences, UC Berkeley, CA, USA
[4] Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{msoeken,drechsle}@cs.uni-bremen.de {sterin,brayton}@berkeley.edu

Abstract—**Reverse engineering is the extraction of word level information from a gate-level netlist. It has applications in formal verification, hardware trust, information recovery, and general technology mapping. A preprocessing step finds blocks in a circuit in which word level components are expected. A second step searches for word level components in these blocks. For this second step, we propose two variants of equivalence checking that consider subfunction containment. We propose algorithms to solve these variants by using subgraph isomorphism. A *simulation graph* (SG) is constructed for the block and for each library component, using a set of permutation-invariant simulation vectors for that component. If a library component SG is a subgraph of the block SG, we have a candidate match, which is then checked by standard equivalence checking. We extend a state-of-the-art subgraph isomorphism algorithm, LAD, to handle simulation graphs efficiently and also propose a SAT-based formulation. Experimental evaluations show that our algorithms can efficiently find 32-bit arithmetic components in blocks with over 300 primary inputs.**

I. Introduction

The problem of reverse engineering (RE) is, given a gate-level netlist, find a word level netlist description which has the same behavior. It can be seen as a generalized technology mapping problem in the following sense. An instance would be to start with a gate-level circuit and a list of word level components, called a library, which contains components such as adders, multipliers, shifters, and other word level components of various bit widths. The goal is to find occurrences of these components in the circuit. While there are many objective functions to be optimized in regular technology mapping, e.g., area, delay, power, clause count in a conjunctive normal form (CNF), and wire count, in RE the single objective is to find all the word level components in the circuit.

RE is of interest for a number of reasons:

1) A register transfer level description (RTL) may not be available; the design may be a legacy without an initial RTL; it may be that an AIG is being passed between various tools with no accompanying RTL, it may have come from bit blasting an RTL and synthesizing it at the bit level.
2) To create a word level description to enable and improve formal verification or synthesis.
3) To understand the structure of an unknown chip.
4) To analyze a chip to help isolate Trojan hardware.

In general, RE is very difficult, but there are many circumstances where the problem is made simpler. The goal of our research is to create set of algorithms that are as efficient as possible and to extend the range of situations where RE is feasible.

RE methodologies typically consist of two main parts: (i) a structural method that decomposes the gate-level circuit into blocks and (ii) a functional method to match sub-circuits of a block with components from a library, assuming that a block's inputs and outputs contain the inputs and outputs of any component to be found.

In this work, we propose an algorithm that targets the second part. Given a block in a circuit and a library, our algorithm finds sub-circuits, called candidates, in the block that have the same simulation behavior, modulo a selected set of simulation vectors, as the component. For this purpose, *simulation graphs* (SGs) of the blocks are created and subgraph isomorphism is applied to match a SG of a component in the library. Once such a candidate has been found it is checked if it is indeed a component using standard combinational equivalence checking. Thus our entire matching algorithm is functional and not structural.

The contributions of this paper are as follows:

1) The problem of finding components in a block of logic is reduced to the subgraph isomorphism problem.
2) An efficient LAD-based subgraph isomorphism algorithm is developed to find matches of component candidates to subgraphs of the block. The requirement that the component outputs be exactly on the block output boundary is relaxed.
3) A symbolic SAT-based subgraph isomorphism algorithm is discussed that is capable of detecting candidates in the presence of inverters at the inputs and outputs in the block.
4) An open source framework is provided with implementations of all algorithms to reproduce the experimental evaluations.

The paper is organized as follows. Relevant definitions and related literature are provided in Sects. II and III, respectively. Sect. IV provides two problem formulations that generalize equivalence checking and discusses how they arise in practice. Sect. V discusses solving such problems by a reduction to subgraph isomorphism of simulation graphs and Sects. VI and VII present implementations based on LAD and SAT, respectively. Sect. VIII discusses an extension that relaxes the assumption that the component's outputs must be contained in the block's outputs. Sect. IX presents experimental results. Finally Sect. X concludes, outlines many future research directions.

II. Preliminaries

General notation and graphs: The notation $[n]$ is a shorthand for the set $\{1, \ldots, n\}$. A *digraph* $G = (V, A)$ consists of a set of vertices V and arcs $A \subseteq V \times V$. Each vertex $v \in V$ has an *in-degree* $d^-(v) = \#\{w \in V \mid (w, v) \in A\}$ and *out-degree* $d^+(v) = \#\{w \in V \mid (v, w) \in A\}$. A digraph is called *$k$-partite* if V can be partitioned into disjoint vertex sets $V_1, \ldots, V_k$ such that there exists no arc $(v, w) \in A$ and no $i \in [k]$ such that $v, w \in V_i$. A k-partite digraph is called *k-layered* if $A \subseteq \bigcup_{i=1}^{k-1} V_i \times V_{i+1}$, i.e., arcs only connect adjacent vertex sets.

A *vertex-labeled* graph also has a set of labels L and a labeling function $l : V \to L$. A *labeled subgraph isomorphism* from $G' = (V', A')$ with labeling function l' to $G = (V, A)$ with labeling function l is an injective function $\mu : V' \hookrightarrow V$ such that $(u, v) \in A' \Leftrightarrow (\mu(u), \mu(v)) \in A$ and $l(u) = l'(\mu(u))$ for all $u \in V'$.

Functions: An n-input m-output Boolean function is an m-tuple of Boolean functions over the variables $x_1, \ldots, x_n$. A Boolean combinational circuit is associated with the function it computes. We will refer to a circuit and the function it computes interchangeably.

Definition 1 (Embedding): An n'-input m'-output Boolean function f' is *embedded* in an n-input m-output Boolean function f if (*1*) there is an injective input-matching function $\pi : [n'] \hookrightarrow [n]$, (*2*) there is an injective output-matching function $\sigma : [m'] \hookrightarrow [m]$ such that $f_{\sigma(j)}(x_1, \ldots, x_n) = f'_j(x_{\pi(1)}, \ldots, x_{\pi(n')})$, for all $j \in [m']$ and $x_1 \ldots, x_n$.

Definition 2 (Simulation vector): A *simulation vector* for a circuit with n inputs is a bitstring of size n. If $(s_1, \ldots, s_n)$ is a simulation vector, then $(f_1(s_1, \ldots, s_n), \ldots, f_m(s_1, \ldots, s_n))$ is the *output of a simulation vector*. The simulation vector is called *k-hot* encoded if exactly k bits are set to one and it is called *k-cold* if exactly k bits are set to zero. Let $S^{\text{hot}}_{n,k}$ and $S^{\text{cold}}_{n,k}$ be the sets of all k-hot and all k-cold simulation vectors, respectively. Note that $\#S^{\text{hot}}_{n,k} = \#S^{\text{cold}}_{n,k} = \binom{n}{k}$ and $S^{\text{hot}}_{n,k} = S^{\text{cold}}_{n,n-k}$.

We also extend the input-matching function from Definition 1 to simulation vectors by padding with zeros where π does not map any inputs of f', i.e., mapping k-hot vectors of length n' to k-hot vectors of length n.

Definition 3: Let $\pi : [n'] \hookrightarrow [n]$ be an input-matching function as above, and let s' be a simulation vector of f'. We define $\pi(s')$ by

$$\pi(s')_j := \begin{cases} s'_i & j = \pi(i), \quad \text{for some } i \\ 0 & \text{(otherwise).} \end{cases}$$

III. Related Work

According to [1] the two main steps to solve a reverse engineering problem are: (1) block identification and (2) matching blocks against components in a library. A component is assumed to be inside a block with its inputs and outputs being contained in the primary inputs and primary outputs of the block. Blocks may contain multiple components and additional logic.

First preliminary approaches have been presented for the first step in [2], [3], however, this step is still considered an open problem with no satisfactory solution proposed so far. One way to circumvent this problem is to have a variety of block matching algorithms for the second step, thereby relaxing the constraints on the blocks. Different approaches for Step 2 make different assumptions on the blocks that are identified by Step 1. Since inputs and outputs of the component are assumed to be primary inputs and primary outputs of the block, all approaches are generalizations of equivalence checking.

In [4], [5], [6], [7], it is assumed that the block neither contains additional logic nor additional inputs and outputs, however, the order of inputs and outputs is unknown. More formally, given two n-input m-output Boolean functions f' and f, the *permutation-independent equivalence checking* (PIEC) problem asks whether f' is embedded (as in Definition 1) in f.

In [8] primary inputs of the block are partitioned into control bits $c_1, \ldots, c_k$, and data bits $x_1, \ldots, x_n$. The order of inputs and outputs is unknown, as well as the values of the control bits that will cause the same functional behavior as that of the component. More formally, given two functions $f(c_1, \ldots, c_k, x_1, \ldots, x_n) = (f_1, \ldots, f_m)$ and $f'(x_1, \ldots, x_n) = (f'_1, \ldots, f'_m)$, the *permutation-independent conditional equivalence checking* (PICEC) problem asks whether there exist two permutations $\pi \in S_n$ and $\sigma \in S_m$ and a propositional function $\psi : [k] \to \mathbb{B}$ such that for all $x_1, \ldots, x_n$ and all $j \in [m]$

$$f_j(\psi(1), .., \psi(k), x_1, \ldots, x_n) = f'_{\sigma(j)}(x_{\pi(1)}, .., x_{\pi(n)}) \quad (1)$$

IV. Problem Formulation and Motivation

1) Problem formulation: We address two problems in this work, called the *subset permutation-independent equivalence checking* (SPIEC) and the *subset negation-and-permutation-independent equivalence checking* (SNPIEC). SPIEC asks whether a smaller function is embedded into a larger one when no input and output correspondence is known. SNPIEC extends the problem and allows inputs and outputs of the larger function to be negated.

The input to the SPIEC problem is an n'-input m'-output Boolean function f' and an n-input m-output Boolean function f. The problem asks whether f' is embedded in f.

The input to the SNPIEC problem is an n'-input m'-output Boolean function f' and an n-input m-output Boolean function f. The problem asks whether there exist two propositional functions

$$p : [n] \to \mathbb{B} \quad \text{and} \quad q : [m] \to \mathbb{B}$$

such that f' is embedded in the function h defined by

$$h_j(x_1, \ldots, x_n) = q(j) \oplus f(p(1) \oplus x_1, \ldots, p(n) \oplus x_n),$$

i.e., the function h is f whose i-th input is inverted if $p(i) = 1$ and j-th output is inverted if $q(j) = 1$. The SNPIEC problem detects a subfunction in a block in the presence of misplaced inverters at the primary inputs and outputs of the block.

2) Motivation: We exploit in our algorithms the fact that many components of interest can have a uniquely characteristic input/output behavior even for a small set of simulation vectors.

As an example, let $f : \mathbb{B}^{2n} \to \mathbb{B}^{n+1}$ be the function of an n-bit adder for which unknown permutations have been

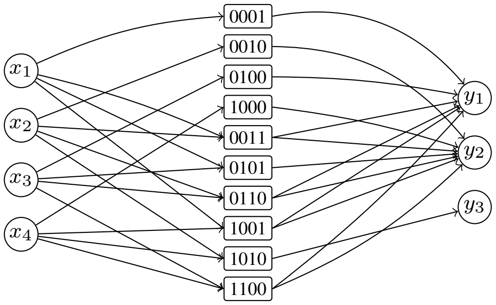


Fig. 1. A Simulation graph of a 2-bit adder. Simulation nodes are annotated with the simulation vector associated with it for convenience.

applied to the inputs and outputs. All one-hot and all two-hot simulation vectors are sufficient to find these permutations using the following arguments. From addition with 0, we know there are one-hot simulation vectors s and s', $s \neq s'$, with $s_i = 1$ and $s_j' = 1$, such that $f(s) = f(s') = y$ is also one-hot with $y_k = 1$. Therefore, i, j, and k refer to the same bit position p in the operands and the output of the adder. To determine the next position $p+1$, the two-hot simulation vector $s \,|\, s'$ ('$|$' refers to bitwise OR) yields a one-hot output value $y' = f(s \,|\, s')$ with $y_l' = 1$. Therefore, $l = p + 1$.

Note that since binary addition is bit-wise commutative, i.e., a_i and b_i for any i can be swapped without changing the result $a + b$; we cannot uniquely determine the partition of input bits into a left-hand operand and a right-hand operand. For other components, such as a multiplier, a different set of simulation vectors is needed. Additionally each component would have its own sequence of deductions for determining the permutations. Therefore, we seek a method that does not depend on the type of component to be found.

V. Reduction to Subgraph Isomorphism

This section describes how to find components in blocks by reducing it to a labeled subgraph isomorphism problem. The key to our approach is the concept of a simulation graph (SG), which captures the behavior of a circuit on a set of simulation vectors.

Definition 4 (Simulation Graph): Let f be an n-input m-output Boolean function and let $S = \{s^{(1)}, \dots, s^{(t)}\}$ be a set of simulation vectors for f. The *simulation graph* G_f^S is a 3-layered vertex-labeled directed graph $G = (X \cup S \cup Y, A_1 \cup A_2)$ where (*1*) $X = \{x_1, \dots, x_n\}$, (*2*) $Y = \{f_1, \dots, f_m\}$, (*3*) $(x_i, s^{(j)}) \in A_1 \Leftrightarrow s_i^{(j)} = 1$, and (*4*) $(s^{(j)}, f_r) \in A_2 \Leftrightarrow f_r(s^{(j)}) = 1$. Labeling is defined by (*1*) $L = \{\mathbf{PI}, \mathbf{PO}\} \cup \mathbb{N}$, (*2*) $l(x_i) = \mathbf{PI}$, (*3*) $l(f_r) = \mathbf{PO}$, and (*4*) $l(s^{(j)}) =$ number of 1s in $s^{(j)}$.

In other words, in an SG, a simulation vector is connected to the inputs that are 1 on it, and to the outputs where it produces a 1. The labels denote the types of node in the graph, inputs, outputs, and the number of ones in a simulation vector.

Example 1: Fig. 1 shows an SG for a 2-bit adder. It has four vertices for the inputs and three vertices for the outputs. Simulation vectors used are all the one-hot and two-hot vectors, resulting in 10 vertices for the simulation vectors. Nodes in the figure are annotated for readability and are not the labels used in the algorithm.

For the remainder of this section, f is an n-input m-output Boolean function, f' is an n'-input and m'-output Boolean function, $K \subseteq \{0, \dots, n'\}$, $S = \bigcup_{k \in K} S_{n,k}^{\mathrm{h}}$ and $S' = \bigcup_{k \in K} S_{n',k}^{\mathrm{h}}$. Let $G_{f'}^{S'}$ denote the SG for f' using S' and let G_f^S denote the SG for f using S. We chose these sets of simulation vectors because of the following property.

Proposition 1: For all $K \subseteq \{0, \dots, n\}$ the set $\bigcup_{k \in K} S_{n,k}^{\mathrm{h}}$ is closed under permutations of the order of the inputs.

Theorem 1: If f' is embedded in f, then there exists a labeled subgraph isomorphism from $G_{f'}^{S'}$ to G_f^S.

Proof: We can use the input and output matching functions of the embedding π and σ to construct a mapping μ from $G_{f'}^{S'}$ to G_f^S:

1) $\mu(x_i') := x_{\pi(i)}$,
2) $\mu(f_r') := f_{\sigma(r)}$,
3) $\mu(s'^{(j)}) := \pi(s'^{(j)})$.

Note that $s'^{(j)}$ is a k-hot vector if and only if $\pi(s'^{(j)})$ is k-hot. Since S is closed under permutation, $\pi(s'^{(j)})$ is a simulation vector node in G_f^S. This mapping preserves the labeling because it maps inputs to inputs, outputs to outputs, and k-hot simulation vectors to k-hot simulation vectors. To show that μ is a subgraph isomorphism observe that

1) $(x_i', s'^{(j)}) \in A_1' \Leftrightarrow (s'^{(j)})_i = 1 \Leftrightarrow \mu(s'^{(j)})_{\pi(i)} = 1 \Leftrightarrow (\mu(x_i'), \mu(s'^{(j)})) \in A_1$, and
2) $(s'^{(j)}, f_r') \in A_2' \Leftrightarrow f_r'(s'^{(j)}) = 1 \Leftrightarrow f_{\sigma(r)}(\mu(s'^{(j)})) = 1 \Leftrightarrow (\mu(s'^{(j)}), \mu(f_r')) \in A_2$ $\blacksquare$

Because a simulation graph is constructed using only a small subset of all possible simulation vectors, the converse does not hold. Instead we can just state the obvious result that if there is a labeled subgraph isomorphism from $G_{f'}^{S'}$ to G_f^S, then f has a subset of inputs and outputs, that behaves like f' on the set of vectors used to construct the simulation graphs.

The existence of a subgraph isomorphism as stated in Theorem 1 depends on the set of simulation vectors used being closed under permutation. Consequently, a single simulation vector can only be supported by the proposed approach if *all* its permutations are considered.

Finding candidate components: Theorem 1 supports an algorithm for finding candidate components in the block. Using the same types of k-hot simulation vectors, we construct a simulation graph for the block, called the *target graph*, and a simulation graph for the component, called the *pattern graph*. If there is a labeled subgraph isomorphism from the pattern graph to the target graph, the mapping of inputs and outputs of the pattern graph can be used to extract a subcircuit of the target graph for use as a candidate for equivalence checking (CEC). If no labeled subgraph isomorphism is found, we conclude from Theorem 1 that the component is not embedded in the block. Note that the subgraph isomorphism problem is NP-complete [9].

Candidate quality: The quality of the candidate, or the likelihood that a candidate is definitely the component, depends on the set of simulation vectors used to construct the simulation graphs.

For example, using just the 0-hot vector, almost guarantees a false positive, and on the adder example, using just the 1-hot vectors does not differentiate between the MSB and LSB.

For each component we can use a simple criteria to judge the quality of candidate sub-blocks generated when using a specific set of simulation vectors; if component f' has its inputs and outputs permuted to create f, does any candidate isomorphism between the corresponding SGs derived from the simulation set, provide a correct matching of the inputs, up to symmetries in f'.

Note that only the matching of the inputs is mentioned. Once the inputs are matched correctly, it becomes easy to match the outputs using random simulation or formal techniques.

k-cold Simulation vectors: The quality of a candidates is likely to be higher if more simulation vectors are used. However, because of the sheer number of k-hot vectors, it becomes impractical to construct the graph beyond 2-hot simulation vectors.

It is tempting to also use n-hot, $(n'-1)$-hot or $(n'-2)$-hot simulation vectors, but if $n > n'$, the size of the target graph may still be very large.

For example, if $n' = 10$ and $n = 100$, the number of $(n'-1) = 9$-hot simulation vectors is just $\binom{10}{9} = 9$, but at the block it is $\binom{100}{9}$ which is impractically huge.

A simple alternative is to use k-cold vectors. We can generalize the definition of a simulation graph to allow k-cold vectors. An input vertex will be connected to a k-cold vector if the input is 0 on that vector. A *k-cold* label on the vectors is used to distinguish them from k-hot vectors.

For simplicity, in this paper we only formally defined and proved results for the k-hot vectors.

Our experimental results demonstrate that 1-hot, 2-hot, and 0-cold simulation vectors were enough to detect many common components, with the exception of multipliers for which 2-hot, 0-cold, and 1-cold simulation vectors were needed.

VI. LAD-based Approach

This section describes an algorithm to solve SPIEC that checks for subgraph isomorphism in SGs using a state-of-the-art algorithm LAD [10], [11]. LAD works on general graphs and does not take the special structure of SGs into account. It is implemented in terms of a constraint satisfaction framework which starts by assigning each vertex $u \in V_P$ a *domain* $D_u \subseteq V_T$ that contains possible candidates for node matching. These domains are refined in the search process using several filtering techniques until either inconsistencies are found, indicated by an empty domain, or no further refinement is possible.

To speed up the search process, it is important to keep the sizes of the domains small at each step. There are two possibilities to reduce the sizes: (i) when initializing the problem and (ii) by using implications during the search process. The latter is more difficult to implement. LAD offers two methods to decrease the domains initially. First, the in-degree $d^-(u)$ and out-degree $d^+(u)$ of a vertex u in the pattern graph cannot exceed the in-degree and out-degree of vertices in the target graph. LAD also supports labeled subgraph isomorphism using a labeling function l. Based on these observations the initial domain for a vertex $u \in V_P$ is

$$\begin{aligned} D_u = \{&v \in V_T \mid d^-(v) \geq d^-(u) \wedge d^+(v) \geq d^+(u)\} \\ &\cap \{v \in V_T \mid l(v) = l(u)\}. \end{aligned} \tag{2}$$

Extending LAD-based subgraph isomorphism: We restrict the domains further by extracting information from the underlying circuits of the SGs. Given a circuit and a primary output u we define $\operatorname{supp}_s(u)$ to be the *structural support* of u, i.e., the set of primary inputs that are reachable from the outputs in a backwards traversal of the circuit starting at u. The *functional support* of u, denoted $\operatorname{supp}(u)$, is the set of primary inputs on which the function represented by u depends. Clearly $\operatorname{supp}(u) \subseteq \operatorname{supp}_s(u)$, i.e., the structural support over-approximates the functional support. Matching output vertices must have the same functional support size and therefore the following constraint is added to Eq. (2) for all $u \in Y_P$:

$$D_u = \ldots \cap \{v \in Y_T \mid \#\operatorname{supp}(v) = \#\operatorname{supp}(u)\} \tag{3}$$

If computing the functional support for the target circuit is too inefficient, one can also use the structural support for a weaker constraint, i.e.,

$$D_u = \ldots \cap \{v \in Y_T \mid \#\operatorname{supp}_s(v) \geq \#\operatorname{supp}(u)\}. \tag{4}$$

These constraints only take the size of the support into account but not the actual inputs in the support. If the functional support is computed, one can add additional so-called *support arcs* (x_i, u_r) to the simulation graph, if and only if $x_i \in \operatorname{supp}(u_r)$. The experimental evaluation will show that these arcs can lead to an improvement of the run-time.

Further improvement can be achieved by making use of *simulation signatures*. There are $\binom{n}{k}$ k-hot and k-cold simulation vectors for circuits with n primary inputs. While simulation can be performed efficiently for small k, their explicit representation as vertices in the SG causes a significant degradation in the run-time for subgraph isomorphism. It is therefore of interest to only include the most effective simulation vectors for SGs. But simulation results of other simulation vectors can still be used in other ways.

We compute for each output u a *simulation signature*, which in our experiments, is a tuple containing the number of 0,1,2-hot, and 0,1,2-cold simulation vectors that drive u to 1. More formally, the simulation signature of an output u in a circuit with n primary inputs is a tuple of values

$$\operatorname{sig}_{n,k}(u) = \#\{x \in S^{\text{hot}}_{n,k} \mid u(x) = 1\}$$

The definitions are analogous for k-cold simulation vectors.

This requires a notion of signature compatibility. Two simulation signatures of a target output $u \in Y_T$ and $u' \in Y_P$ are compatible, denoted $\operatorname{sigcomp}(u, u')$, if and only if for all values in the tuple

$$\operatorname{sig}_{n,k}(u) = \sum_{i=0}^{k} \binom{n-n'}{i} \cdot \operatorname{sig}_{n',k-i}(u') \tag{5}$$

holds. As one instance of this equation, we have

$$\operatorname{sig}_{n,1}(u) = \operatorname{sig}_{n',1}(u') + (n-n') \cdot \operatorname{sig}_{n',0}(u),$$

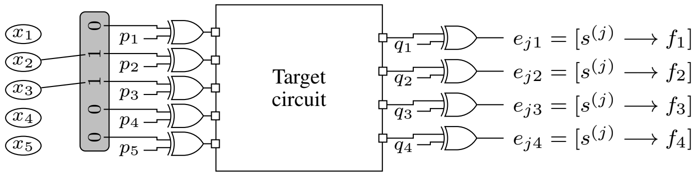

x_1
x_2
x_3
x_4
x_5
0
1
1
0
0
p_1
p_2
p_3
p_4
p_5
Target circuit
q_1
q_2
q_3
q_4
$e_{j1} = [s^{(j)} \longrightarrow f_1]$
$e_{j2} = [s^{(j)} \longrightarrow f_2]$
$e_{j3} = [s^{(j)} \longrightarrow f_3]$
$e_{j4} = [s^{(j)} \longrightarrow f_4]$

Fig. 2. Handling inverters in SAT-based subgraph isomorphism (circuit construction for simulation vector 00110)

i.e., the size of the on-set for one-hot encoded simulation vectors in the target graph, is the sum of $\operatorname{sig}_{n',1}(u')$ (considering the '1' is assigned to one of the n' matching inputs) and $(n-n')\operatorname{sig}_{n',0}(u)$ (considering the '1' is assigned to one of the $n-n'$ non-matching inputs).

Simulation signatures further restrict the domains:

$$D_u = \ldots \cap \{v \in Y_T \mid \operatorname{sigcomp}(u,v)\} \tag{6}$$

VII. SAT-based Approach

We introduce a SAT-based subgraph isomorphism based on the formulation given in [12] mainly for two reasons. First, to show the dominance of the LAD-based approach in terms of scalability (see experimental results in Sect. IX). Second, a symbolic representation of SGs makes it easier to solve the SNPIEC problem (where possible inverters are at the inputs and outputs). The presence of possible inverters affects the target graph such that it cannot be represented explicitly, which precludes application of the LAD-based approach.

The formulation as a SAT instance is inspired by the formulations described in [12], [13], [14]. The SAT formulation includes the same optimization techniques (blocking using labels, structural, functional support, support arcs, and simulation signatures) and additionally exploits symmetry breaking based on input symmetries. Our LAD algorithm does not consider the latter because it is more complicated and requires several nontrivial changes throughout the whole algorithm. Due to space limitations, we only focus on the extension of the formulation to solve SNPIEC problems.

SAT based formulation of SNPIEC: If inverters are possibly present at inputs and outputs (SNPIEC problem) in the target circuit, simulation results are changed and therefore also the target graph. This seems to prohibit the application of subgraph isomorphism algorithms to find candidate components. However, a SAT based formulation can be made for SNPIEC.

The presence of inverters at inputs and outputs changes the function values of the target circuits and causes arcs between simulation vectors in S_T and outputs in Y_T in the target graph, which cannot be determined explicitly. To formulate SNPIEC the presence of arcs must be determined symbolically based on additional polarity variables $p_1, \ldots, p_n$ and $q_1, \ldots, q_m$ with $n = \#X_T$ and $m = \#Y_T$. One way to determine the polarity variables is to construct a circuit for each simulation vector as illustrated in Fig. 2 that contains variables e_{jr} which represent an arc between simulation vertex $s^{(j)}$ and output f_r in the target graph. In this case there are 5 inputs and 4 outputs and the simulation vector is 00110. As can be seen, the input vertices and simulation vector vertices can be connected explicitly. For each input and output of the target circuit, an XOR gate is added that is controlled by a polarity variable. Besides the polarity variables, inputs to the XOR gates are the simulation bits and the original outputs. The new outputs of the circuit are the symbolic values for e_{jr}. This circuit is copied for *each* simulation vector, which results in a very large circuit with $m+n$ primary inputs (the polarity variables) and $m \cdot \#S_T$ primary outputs (the number of e_{jr} variables). This large circuit is transformed into a CNF formula and added to the SAT formulation. In order to get a smaller formula, we optimized the circuit in our experiments using ABC [15] before translating it into a CNF. However, this approach is not tractable (experiments showed reasonable runtimes only for small instances) and further research is needed if motivated.

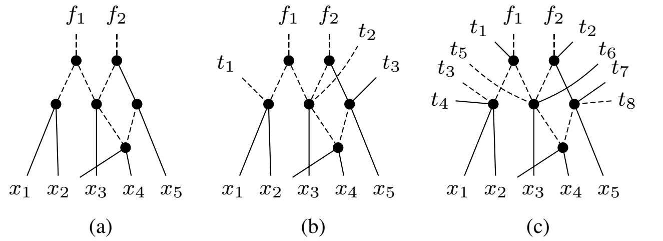


Fig. 3. Output feathering: (a) original circuit, (b) feathering with respecting edge polarities, and (c) feathering all polarities. Dashed edges are inverterd.

VIII. Relaxing the Constraints of Block Identification

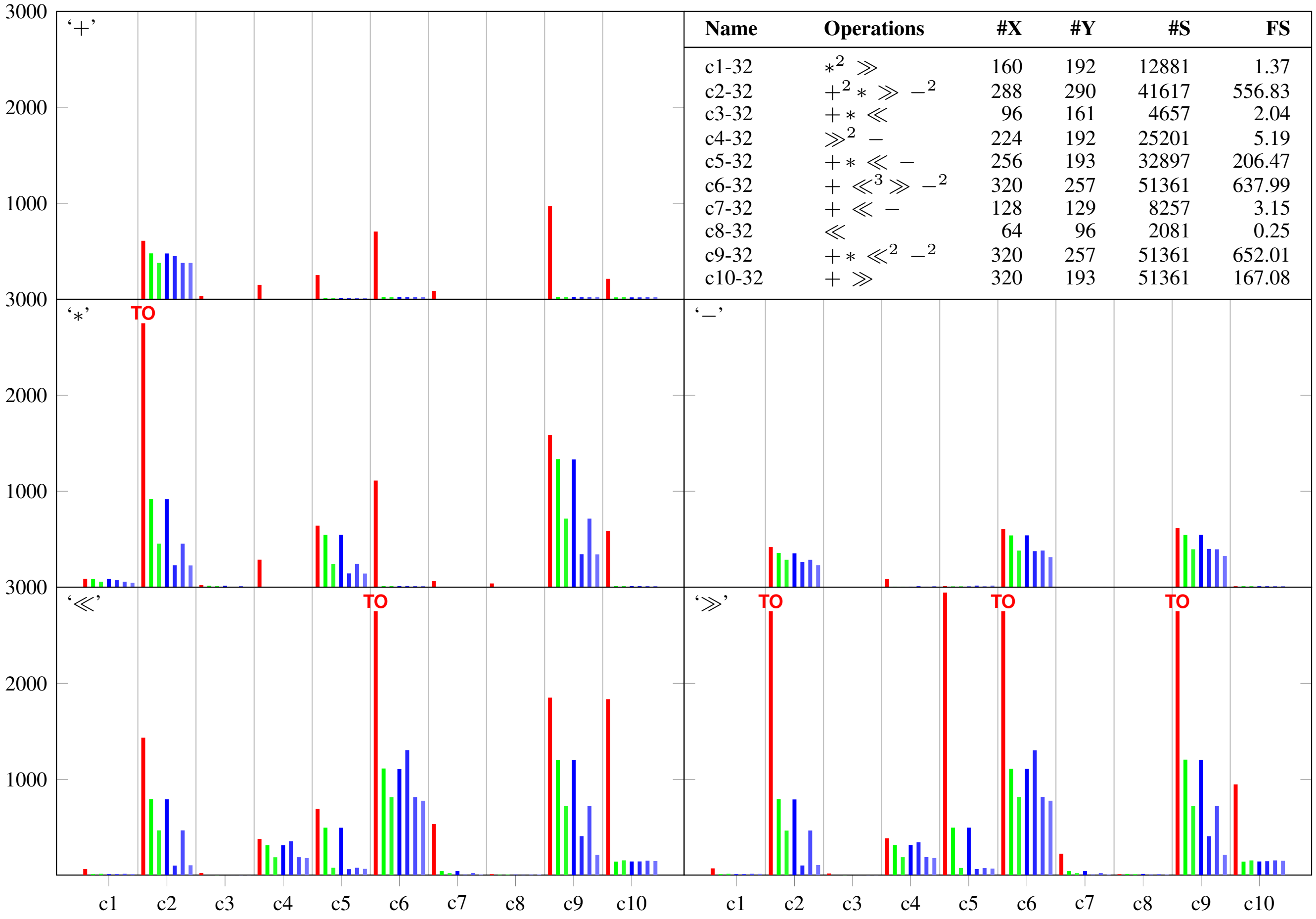
All discussed generalized equivalence checking problems assume that the primary inputs and primary outputs of the component f' are also primary inputs and primary outputs of the block f. Relaxing this assumption can help in the block identification problem. We propose a technique called *output feathering* as a preprocessing step to a SPIEC solver for this. Note that output feathering exploits the subset relation in the problem definition of SPIEC and is therefore not applicable to the other block matching algorithms that were described in Sect. III.

Output feathering first levelizes the circuit—in our case an AIG—and then creates outputs for each node in the k topmost levels. We allow two modes of output feathering. The first creates outputs according to the polarities of outgoing edges whereas the second mode creates an output and its negation for each node. Fig. 3 illustrates output feathering and both modes. This is practical because our SG based method is quite insensitive to the number of outputs of the block.

IX. Experimental Evaluation

We implemented the approaches, discussed in this paper, in C++ and present our evaluations in this section.[1] We implemented a tool (part of the above mentioned source code) that generates gate-level circuits meeting the assumptions on the blocks our algorithm expects from block identification. The tool randomly chooses from multiple arithmetic components in a

[1] The implementation is called 'find_subcircuit.' The source code and all benchmarks are available at `http://www.informatik.uni-bremen.de/~msoeken/revenge-1.0.tar.gz`.



Name	Operations	#X	#Y	#S	FS
c1-32	$*^2 \gg$	160	192	12881	1.37
c2-32	$+^2 * \gg -^2$	288	290	41617	556.83
c3-32	$+ * \ll$	96	161	4657	2.04
c4-32	$\gg^2 -$	224	192	25201	5.19
c5-32	$+ * \ll -$	256	193	32897	206.47
c6-32	$+ \ll^3 \gg -^2$	320	257	51361	637.99
c7-32	$+ \ll -$	128	129	8257	3.15
c8-32	$\ll$	64	96	2081	0.25
c9-32	$+ * \ll^2 -^2$	320	257	51361	652.01
c10-32	$+ \gg$	320	193	51361	167.08

Fig. 4. Experimental evaluation for the LAD-based approach. For each circuit and for each component, the seven bars (shown from left to right) represent the LAD algorithm with different sets of enhancements: (i) default LAD, (ii) structural support, (iii) structural support + simulation signatures, (iv) functional support, (v) functional support + simulation signatures, (vi) functional support + support arcs, and (vii) functional support + support arcs + simulation signatures

library as well as additional components not in the library (e.g., wide AND and OR gates) to create "noise" in the circuit. Then, inputs are randomly added and connected to the components; components may share common inputs; there may be several instances of the same component. The library used in the experiments consisted of an adder ($+$), multiplier ($*$), left-shifter ($\ll$), right-shifter ($\gg$), and a subtracter ($-$). These were mapped to gate level circuits for different bit widths. The experiments were carried out on an Intel Xeon processor with 2.4 GHz, 64 GB RAM, running Linux 3.17. Run-times are given in seconds with a timeout (TO) of 3600 seconds. The sets of experiments done, (i) show that the LAD-based approach is efficient for solving SPIEC problems, (ii) compare the LAD-based approach to the SAT-based approach, (iii) show how the SAT-based approach scales for SNPIEC problems, and (iv) use our approach to solve PICEC problems and compare with the state-of-the-art.

LAD-based approach: The results are listed in Fig. 4. The table in the top right lists properties of the 10 circuits used in the experiments. The first column lists the identifier of each circuit (c1 – c10). Each circuit in the library has 32-bit primary inputs. The second column lists which arithmetic operations are contained in the circuit and superscripts denote the number of instances of the operator. If no superscript is specified, it occurs only once. The columns $\#X$, $\#Y$, and $\#S$ give the numbers of inputs, outputs, and simulation vectors of the target graph. The sum of these three numbers is the number of nodes in the target graph. For the experiments, we used all-hot, one-hot, and two-hot simulation vectors for all components except the multiplier, for which we used one-cold instead of one-hot simulation vectors, since multiplication by 0 creates no arcs between simulation vertices and output vertices. We used the ABC [15] command '`print_supp`' which uses a simple method to compute the functional support. The run-time to compute the functional support is given in column *FS*. These numbers can be improved considerably, however, note that the functional support needs to be computed only once for each circuit.

The rest of the figure consists of blocks of plots for each component in the library. The component being matched is illustrated by its operator symbol in the top left corner. Each block of plots is separated into 10 compartments, one for each circuit, and each compartment has seven bars that show the run-times respectively for the following configurations of the

LAD-based approach, from left to right:

(i) A modification of the original LAD approach for subgraph isomorphism from [10]. This implementation considers domain constraints for vertex degrees and vertex labels as described in Eq. (2), referred to as LAD.[2]

(ii) LAD with structural support for domain constraints (see Eq. (4)), referred to as LAD_s.

(iii) LAD_s + simulation signatures.

(iv) LAD with functional support for domain constraints (see Eq. (3)), referred to as LAD_f.

(v) LAD_f + simulation signatures.

(vi) LAD_f + support arcs.

(vii) LAD_f + support arcs + simulation signatures.

Note that the approaches only find a candidate mapping but do *not* perform the final CEC equivalence checks. We performed these checks separately using the ABC command '`iprove`' and did not add the run-times to the values in the plots. In fact, for all operations except the multiplier, the equivalence checks could be performed in less than a second. As equivalence checking multipliers is known to be a hard problem, we manually validated the correctness of the computed mapping based on the port names. All matchings determined by the algorithms were correct, but during the initial evaluations, we experienced several wrong matchings for the multiplier, which were resolved once we added the one-cold simulation vectors. This demonstrates that an appropriate set of types of simulation vectors must be chosen individually per component.

The main observations on the experimental results are:

1) When incorporating the support, the run-time is significantly better; in some cases LAD_s and LAD_f can find a matching within a few seconds while LAD does not find a solution within one hour (see, e.g., 'c6-32' and 'c9-32').
2) In the SPIEC problem, a left-shifter is equivalent to a right-shifter because $a \ll b = (a^R \gg b)^R$ where a^R is the reverse of a. This symmetry is evident in the run-times of LAD_s and LAD_f since run-times are not affected by which component is sought. However, in the LAD approach the run-times diverge significantly in some cases (see, e.g., 'c5-32' and 'c7-32').
3) The best performance is achieved for functional support with support arcs and simulation signatures. Often, the support arcs don't make a difference. Also, if neither support arcs nor simulation signatures are used, the difference between structural and function support based domain restriction is marginal.
4) Generally run-times were significantly better when the component was not present (see, e.g., the 32-bit circuits for adders, multipliers, and subtracters).
5) Subtracters and adders are only hard to find if they occur more than once in the block.

SAT-based approach for SPIEC: We scaled down the circuits of the previous experiment to 8 bits and ran the SAT-based approach mentioned in Sect. VII using MiniSAT [16] as the back-end solver, referred to as SAT in the following. Table I lists the results of comparing SAT with LAD_s. Since the instances become extremely large (the number of variables to formalize μ_S is of order $O(|X_P|^4)$), a long run-time is already spent on creating the instance; the solving time makes up approximately 25%. Note that the run-times of the LAD_s approach can be several orders of magnitude faster, e.g., 'c6' and 'c9'. Also the run time of SAT seems to be almost independent of whether the component is contained in the block or not, but is highly correlated with the number of nodes in the SG. Incremental SAT techniques (e.g., with activation literals) may improve the run-times since many considered instances are similar.

TABLE I
EXPERIMENTAL EVALUATION FOR THE SAT-BASED APPROACH TO SOLVE SPIEC

Name	Operations	$+$		$*$		$\ll$		$\gg$		$-$	
		SAT	LAD_s	SAT	LAD_s	SAT	LAD_s	SAT	LAD_s	SAT	LAD_s
c1-8	$*^2 \gg$	7.2	0.0	8.6	0.2	8.6	0.0	7.9	0.0	7.6	0.0
c2-8	$+^2 * \gg -^2$	83.6	0.3	89.8	1.2	89.8	0.7	75.3	0.8	87.4	0.3
c3-8	$+ * \ll$	1.8	0.0	1.9	0.1	1.9	0.0	1.6	0.0	1.7	0.0
c4-8	$\gg^2 -$	22.7	0.0	22.2	0.0	22.2	0.3	29.5	0.3	30.3	0.0
c5-8	$+ * \ll -$	48.3	0.0	52.3	0.7	52.3	0.4	46.6	0.4	50.3	0.0
c6-8	$+ \ll^3 \gg -^2$	123.5	0.0	134.9	0.0	134.9	0.8	131.7	0.8	134.6	0.5
c7-8	$+ \ll -$	4.0	0.0	4.2	0.0	4.2	0.1	3.8	0.1	4.3	0.0
c8-8	$\ll$	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0	0.5	0.0
c9-8	$+ * \ll^2 -^2$	123.3	0.0	155.2	1.3	155.2	0.9	131.8	0.9	134.5	0.4
c10-8	$+ \gg$	120.8	0.0	93.5	0.0	93.5	0.1	117.6	0.1	99.0	0.0

SAT-based approach for SNPIEC: We tried to evaluate the SAT-based approach to solve SNPIEC, denoted $\text{SAT}_\mathbf{n}$, for all circuits of bit width 8; however, no results were obtained within the timeout. The approach is not yet tractable and further research is needed if it turns out that SNPIEC problems occur frequently in RE. One possible direction for future research is the exploitation of exists-forall SAT solvers (see, e.g., [17]) as they are used to solve the PICEC problem in [8].

Comparison to PICEC: In [8], the PICEC problem was solved using a SAT-based method. The input of the problem is partitioned into control and data bits and the aim is to find an assignment of the control bits and a permutation of the data bits. When the number of control bits is small, one can solve such a PICEC problem as a sequence of SPIEC problems by enumerating all control bit assignments, propagating them through the circuit, and stopping once a match has been found. We performed this experiment based on LAD_s on satisfying instances reported in [8] and compared the results with the approach described in [8] with preprocessing, signatures, and Yices [18] as the back-end solver, called PICEC in the following.

TABLE II
SOLVING PICEC WITH SPIEC

Name	#PI	#PO	#C	$+$		$*$		$-$		$\equiv$		$<$	
				LAD_s	PICEC	LAD_s	PICEC	LAD_s	PICEC	LAD_s	PICEC	LAD_s	PICEC
mul8	16	8	0			1.7	1.9						
mul16	32	16	0			TO	N/A						
simple_alu	64	32	2	6.9	361.3			1.4	182.2				
full_alu	64	32	4	55.4	N/A								
fake2670	133	1	5							0.7	0.1	0.1	0.5
c3540	16	22	34	TO	35.5								

[2] The run-time of our LAD implementation is significantly better compared to the original since we replaced an expensive recursive procedure that stores data on a stack by one that stores data on a heap. Further, by using our own implementation, we avoid writing the SGs to temporary files and can work directly on the data structure. Since the improved algorithm is based on this implementation, the comparison of LAD to LAD_s and LAD_f, is more fair.

The results are given in Table II, which lists the name of each circuit, its number of primary inputs and primary outputs, and the number of control bits (not counted in primary inputs). In the original experiments in [8], individual operations were specified for each circuit. These are listed in the last column together with the required run-times. The run-times for LAD$_s$ include the time spent on equivalence checking since this step is included in PICEC. However, note that the partition of input bits is known in PICEC but not in LAD$_s$. For some benchmarks, no results were available (N/A). For benchmark 'mul16' the equivalence check did not terminate within one hour, however, the correct candidate was determined in 0.1 seconds. This shows an advantage of the SPIEC approach being decoupled from the equivalence checker. A different equivalence checker implementation might be able to determine equivalence faster. Benchmark 'c3540' created a too large search space with 34 control inputs, and for such cases PICEC is clearly superior.

X. Conclusions

We presented algorithms for new variants of combinational equivalence checking that integrates into other approaches required for tackling RE problems. They find a component in a larger circuit and a mapping of primary inputs and outputs of the component to the larger circuit. We showed that the problem can be solved efficiently using algorithms for subgraph isomorphism on their SGs, exploiting additional functional information of the circuits to reduce the search space. It was demonstrated that our approach is viable for solving PICEC problems. To solve equivalence checking problem in which the polarity of inputs and outputs may be inverted, we discussed an alternative SAT formulation.

There are many directions for future research. For the experiments in this paper we only considered a fixed set of simulation vector types for each type of component. However, for some components this may be an overkill and the result could have been found by using a smaller set. This seems to suggest that each component should come with its own set of simulation vectors that are known to be sufficient, reducing the sizes of the target and component SGs. However, we should experiment with this to see if it improves run-times. On the other hand, iterative methods can be used to dynamically extend the SGs with k-hot or k-cold vectors for larger k by using learned information to reduce the space of simulation vectors.

Although all matchings were correct in our experiments, it has not been discussed how to proceed if there remain ambiguities in the domains after LAD has terminated. There are several possible scenarios. The LAD-based approach could be extended to yield all possible matchings instead of only the first one. Iterative methods driven by counter examples are another promising solution, however, since all simulation-vectors must be invariant to permutation, counter examples cannot be exploited in a straight-forward manner.

The algorithms presented in this paper do not perform the equivalence check; rather they find a possible matching that can be given to a standard CEC checker. We noted that the run-time required by the equivalence check was negligible compared to the run-time required to determine the mapping, except for multipliers. We propose to investigate the use of structural equivalence checkers for such cases with a set of common multiplier implementations as additional components.

In general, we want to enlarge the large class of library components with its set of simulation types for which an SG-based method can correctly identify sub-isomorphisms. Operators might include square, square root, log, and compositions of various operators like multiply-add.

Acknowledgments: This research is supported by the German Academic Exchange Service (DAAD) in the PPP 57134066 and by the German Federal Ministry of Education and Research (BMBF) in the project SPECifIC under grant 01IW13001. We want to thank NSA for support through contract 'Equivalence checking in crypto-analytic applications' and NSF under contract 1219154.

References

[1] W. Li, Z. Wasson, and S. A. Seshia, "Reverse engineering circuits using behavioral pattern mining," in *Int'l Symp. on Hardware-Oriented Security and Trust*, 2012, pp. 83–88.

[2] W. Li, A. Gascón, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia, "WordRev: finding word-level structures in a sea of bit-level gates," in *Int'l Symp. on Hardware-Oriented Security and Trust*, 2013, pp. 67–74.

[3] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascón, W. Y. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik, "Reverse engineering digital circuits using structural and functional analyses," *IEEE Trans. Emerging Topics Comput.*, vol. 2, no. 1, pp. 63–80, 2014.

[4] J. Mohnke, P. Molitor, and S. Malik, "Establishing latch correspondence for sequential circuits using distinguishing signatures," *Integration*, vol. 27, no. 1, pp. 33–46, 1999.

[5] Y. Lai, S. Sastry, and M. Pedram, "Boolean matching using binary decision diagrams with applications to logic synthesis and verification," in *Int'l Conf. on Computer Design*, 1992, pp. 452–458.

[6] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large Boolean functions with applications to technology mapping," *Formal Methods in System Design*, vol. 10, no. 2/3, pp. 137–148, 1997.

[7] C. Lai, J. R. Jiang, and K. Wang, "Boolean matching of function vectors with strengthened learning," in *Int'l Conf. on Computer-Aided Design*, 2010, pp. 596–601.

[8] A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanović, and S. Malik, "Template-based circuit understanding," in *Formal Methods in Computer-Aided Design*, 2014, pp. 83–90, benchmarks and tools are available at https://bitbucket.org/spramod/fmcad14-experiments.

[9] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[10] C. Solnon, "AllDifferent-based filtering for subgraph isomorphism," *Artif. Intell.*, vol. 174, no. 12-13, pp. 850–864, 2010.

[11] S. Zampelli, Y. Deville, and C. Solnon, "Solving subgraph isomorphism problems with constraint programming," *Constraints*, vol. 15, no. 3, pp. 327–353, 2010.

[12] C. Anton and L. Olson, "Generating satisfiable SAT instances using random subgraph isomorphism," in *Canadian Conference on Artificial Intelligence*, 2009, pp. 16–26.

[13] V. Arvind, P. P. Kurur, and T. C. Vijayaraghavan, "Bounded color multiplicity graph isomorphism is in the #L hierarchy," in *Conf. on Computational Complexity*, 2005, pp. 13–27.

[14] J. Torán, "On the resolution complexity of graph non-isomorphism," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2013, pp. 52–66.

[15] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.

[16] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.

[17] C. M. Wintersteiger, Y. Hamadi, and L. M. de Moura, "Efficiently solving quantified bit-vector formulas," *Formal Methods in System Design*, vol. 42, no. 1, pp. 3–23, 2013.

[18] B. Dutertre, "Yices 2.2," in *Computer Aided Verification*, 2014, pp. 737–744.