Oracle-Guided Incremental SAT Solver User manual

Xiangyu Zhang, Cunxi Yu, Duo Liu, Daniel Holcomb

xiangyuzhang@umass.edu

Software User Manual

version Beta 1.0.0, 20 April 2016

Abstract

This document is the Software User Manual for the Oracle-Guided Incremental SAT Solver (Solver). The Software User Manual instructs how to install and use the Oracle-Guided Incremental SAT Solver (Solver).

Contents

1 Introduction

1.1 Purpose

The Oracle-Guided Incremental SAT Solver (Solver) is specified in solving *$camouflaged$* circuit with extremely high efficiency.

1.2 Problem formulation

The *$camouflaged$* circuit model represents the reverse engineer's uncertainty about the logic gates. The *oracle* represents the real physical object, where the user has only ability to apply inputs and observe outputs, but cannot look inside to see what the gates are. The *Solver* will use the *oracle* as guide to solve for the logic function of the *$camouflaged$* circuit.

1.3 Principle

Solver executes a loop that continually finds new input and output vectors using SAT queries and an oracle circuit model. After some number of iterations, the constraints accumulated are sufficient to rule out all logical functions except for the one that is the true function of the obfuscated circuit. Detailed description please refer to our DATE paper [1].

1.4 Terminology

- Oracle program: an executable program that can produce the correct circuit output for any input that is provided to it via file PI.txt and generate corresponding PO.txt.
- Camouflaged circuit: obfuscated *oracle* circuit.
- Allowed values: Allowed values are the sets of values that the programming bits for each obfuscated component can take. Given that the programming bits select the functionality of the circuit, the combinations of the allowed values for all components represents the space of hypotheses for

the overall circuit function. Finding specific values from these choices is the deobfuscation problem, and is the goal of our program.

2 Tutorial

2.1 Dependencies

NOTE: The *Solver* is based on MINISAT.

- MINISAT module: modified from original MINISAT
  - (MROOT)/core: includes MINISAT solver head file, implementation file and related supporting file.
  - (MROOT)/mtl: includes MINISAT templates and make file.
  - (MROOT)/utils: includes MINISAT utilities.
- Solver module: solver module
  - (MROOT)/simp: includes *Solver* main file and MINISAT Simp-Solver sourcce file.
  - (MROOT)/incre: includes *Solver* source file and all the related utilities.
  - (MROOT)/Oracle: includes sample *Oracle* and sample Shell script.

2.2 Installation

Makefile is included in (MROOT)/simp directory, change to MROOT directory and use command below to install

1. *$ export MROOT=(solver-dir)*
2. *$ cd simp*
3. *$ make rs*
4. *$ cp SOLVER_static (install-dir)/SOLVER*

NOTE:

1. The minimum requirement for complier is **g++ 4.9**.
2. If make fail, use **$ make clean** and try again.

2.3 Command line usage

After installation, *solver* can be accessed from command line:

```
$ SOLVER [options] <Cam.v > <Orac.sh >
```

- Cam.v : the verilog netlist of the circuit to be deobfuscated. The netlist must have the necessary annotations for the PIs and annotations to define the allowed values for the programming bits. Note that only a restricted subset of Verilog can be used, as defined below.
- Orac.sh: shell script to query *Oracle*.
- -d, - -debug: change to debug mode, solver will generate log message and log files.
- -o, - -outfile: export solution to this file .

For example, if the *oracle* shell is "c432-Oracle.sh" and the *camouflaged* circuit is "c432-mux4-101.v", then the command can be:

```
$ SOLVER -d - -outfile Solution.txt c432-mux4-101.v c432-Oracle.sh
```

2.4 Oracle Description

During the solving, *Solver* will generate 'PI.txt' file and read 'PO.txt' file. For stability sake, the filename can not be changed. So please make sure your *Oracle* will can read 'PI.txt' and export 'PO.txt' in working directory. Sample PO.txt is in folder *Oracle*. The first line are the signal names, and the second line are the signal values. Each netname or value in PO file is seperated by a tab, each line in PO file is seperated by a line break. PI file uses the same format. For example:

```
N1(\t)N2(\t)N3(\t)N4(\t)N5(\t)CONST1(\t)CONST0(\n)
1(\t)1(\t)1(\t)0(\t)1(\t)1(\t)0(\n)
```

The user's oracle program could produce the outputs by any means they desire, such as by physically querying the obfuscated circuit via its scan chain. For sake of evaluation, users may wish to implement the Oracle by running simulation of a non-obfuscated circuit function, or querying a pre-programmed exhaustive look-up table of PI-PO pairs. In the example oracle files we provide, the PIs are mapped to POs by evaluating the circuit CNF using Minisat with appropriate inputs applied.

2.5 Sample Camouflage format

Solver takes gate-level verilog, the following is an example:

```
module  c17 (N1,N2,N3,N4,N5,N10,N11,CONST1,CONST0,D_0,D_1,D_2,D_3);
input  N1,N2,N3,N4,N5,CONST1,CONST0 ;//RE__PI;
input D_0,D_1 ;//RE__ALLOW(00,01,10,11);
input D_2,D_3 ;//RE__ALLOW(00,01,10,11);
output N10,N11;
wire N6,N7,N8,N9,D_0_NOT,D_1_NOT,N7_NOT,N7_OBF,ED_0,ED_1,ED_2,ED_3,ED_4
    ,ED_5,ED_6,ED_7,ED_8,ED_9,D_2_NOT,D_3_NOT,N6_NOT,N6_OBF,ED_10,ED_11
    ,ED_12,ED_13,ED_14,ED_15,ED_16,ED_17,ED_18,ED_19;
nand2 gate1( .a(N1), .b(N3), .O(N6) );
nand2 gate2( .a(N3), .b(N4), .O(N8) );
nand2 gate3( .a(N2), .b(N8), .O(N7) );
nand2 gate4( .a(N8), .b(N5), .O(N9) );
nand2 gate5( .a(N6_OBF), .b(N7), .O(N10) );
nand2 gate6( .a(N7_OBF), .b(N9), .O(N11) );
inv1 gate7( .a(D_0), .O(D_0_NOT) );
inv1 gate8( .a(D_1), .O(D_1_NOT) );
inv1 gate9( .a(N7), .O(N7_NOT) );
and2 gate10( .a(N7), .b(D_0_NOT), .O(ED_0) );
and2 gate11( .a(N7_NOT), .b(D_0_NOT), .O(ED_1) );
and2 gate12( .a(CONST1), .b(D_0), .O(ED_2) );
and2 gate13( .a(CONST0), .b(D_0), .O(ED_3) );
and2 gate14( .a(ED_0), .b(D_1_NOT), .O(ED_9) );
```

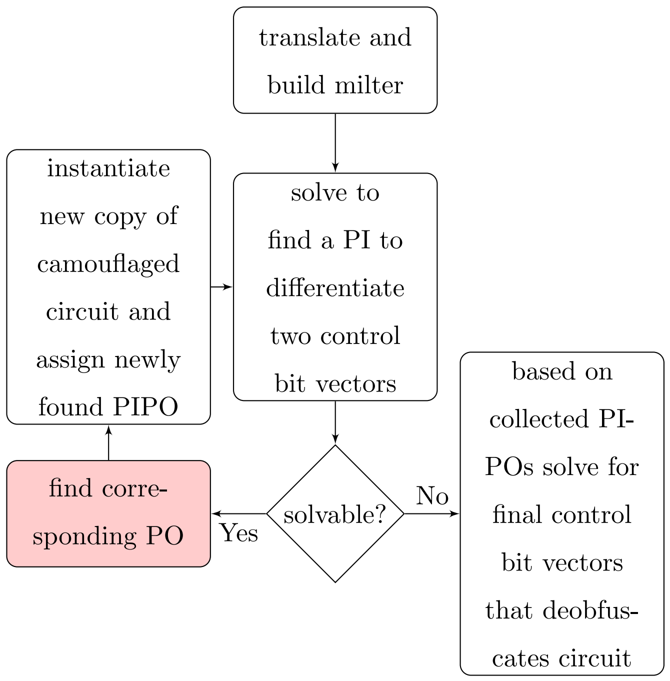
```
and2 gate15( .a(ED_1), .b(D_1), .O(ED_7) );
and2 gate16( .a(ED_2), .b(D_1_NOT), .O(ED_5) );
and2 gate17( .a(ED_3), .b(D_1), .O(ED_4) );
or2  gate18( .a(ED_4), .b(ED_5), .O(ED_6) );
or2  gate19( .a(ED_6), .b(ED_7), .O(ED_8) );
or2  gate20( .a(ED_9), .b(ED_8), .O(N7_OBF) );
inv1 gate21( .a(D_2), .O(D_2_NOT) );
inv1 gate22( .a(D_3), .O(D_3_NOT) );
inv1 gate23( .a(N6), .O(N6_NOT) );
and2 gate24( .a(N6), .b(D_2_NOT), .O(ED_10) );
and2 gate25( .a(N6_NOT), .b(D_2_NOT), .O(ED_11) );
and2 gate26( .a(CONST1), .b(D_2), .O(ED_12) );
and2 gate27( .a(CONST0), .b(D_2), .O(ED_13) );
and2 gate28( .a(ED_10), .b(D_3_NOT), .O(ED_19) );
and2 gate29( .a(ED_11), .b(D_3), .O(ED_17) );
and2 gate30( .a(ED_12), .b(D_3_NOT), .O(ED_15) );
and2 gate31( .a(ED_13), .b(D_3), .O(ED_14) );
or2  gate32( .a(ED_14), .b(ED_15), .O(ED_16) );
or2  gate33( .a(ED_16), .b(ED_17), .O(ED_18) );
or2  gate34( .a(ED_19), .b(ED_18), .O(N6_OBF) );
endmodule
```

There are several points that need attention regarding the Verilog netlist:

- the line declaring primary inputs should be followed by "//RE__PI;".
- the line declaring control bits should be noted by "//RE__ALLOW();".
- allowed values for each group of control bits should be written inside paretheses of "//RE__ALLOW();", for example "//RE__ALLOW(1,0);". If one camouflaged gate requires more than one control bits, for example it needs two bits, use the format "//RE__ALLOW(00,01,10,11);".
- the number of control bits in an input line must be equal to the length of allowed values after it. For example, "input D_1; RE__ALLOW(00,01,10,11);" is illegal because it defines a single programming bit but describes a set of 2-bit values for the bit to take.

- primary inputs named CONST1 and CONST0 will be interpreted as values 1 and 0.
- solver can accept the following kinds of gate:
  - inv
  - and (with any number of fanin)
  - or (with any number of fanin)
  - xor
  - nor (with any number of fanin)
  - nand (with any number of fanin)
  - buf

2.6 Flow diagram


translate and build milter
solve to find a PI to differentiate two control bit vectors
instantiate new copy of camouflaged circuit and assign newly found PIPO
find corresponding PO
solvable?
Yes
No
based on collected PI-POs solve for final control bit vectors that deobfuscates circuit

NOTE: The while filled blocks are SOLVER tasks. And the red filled block belongs to user defined oracle.

3 History Version

- version Beta 1.0.0, 20 April 2016

4 Contact

Xiangyu Zhang: xiangyuzhang@umass.edu

5 Reference

1. *Oracle-Guided Incremental SAT Solving to Reverse Engineer Camouflaged Logic Circuits*, **DATE 2016**