Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Symbolic execution based test-patterns generation algorithm for hardware Trojan detection

Lixiang Shen[a,b,*], Dejun Mu[a], Guo Cao[c,d], Maoyuan Qin[a], Jeremy Blackstone[e], Ryan Kastner[e]

[a] School of Automation, Northwestern Polytechnical University, Xian 710072, PR China
[b] School of Computer Information and Engineering, Changzhou Institute of Technology, Changzhou 213032, Jiangsu, PR China
[c] School of Management, Northwestern Polytechnical University, Xian 710072, PR China
[d] School of Economics and Management, Changzhou Institute of Technology, Changzhou 213032, Jiangsu, PR China
[e] Department of Computer Science and Engineering at the University of California, San Diego, CA 92093-0404, United States

ARTICLE INFO

Article history:
Received 10 January 2018
Revised 1 June 2018
Accepted 9 July 2018
Available online 17 July 2018

Keywords:
Hardware Trojan
Symbolic execution
Satisfiability modulo theory
Metamorphic testing
Control flow graph

ABSTRACT

Hardware Trojan detection is a very difficult challenge. However, the combination of symbolic execution and metamorphic testing is useful for detecting hardware Trojans in Verilog code. In this paper, symbolic execution and metamorphic testing were combined to detect internal conditionally triggered hardware Trojans in the register-transfer level design. First, control flow graphs of Verilog code were generated. Next, parallel symbolic execution and satisfiability modulo theories solver generated test patterns. Finally, metamorphic testing detected the hardware Trojans. The work used Trust-Hub benchmarks in experiments.

1. Introduction

As modern embedded system design becomes more complex, malicious insiders have more opportunities to modify the hardware with hardware Trojans. Hardware Trojans are a type of malicious code that cause insertions, deletions and modifications to the original hardware design. They can threaten integrity, confidentiality and availability by altering the original function of the design, leaking sensitive information (Jin and Makris, 2010) and reducing the reliability of the hardware.

Hardware Trojans can cause very serious security problems (Li et al., 2016) in many industries. Nissim et al. (2017) described several USB hardware Trojans which installed backdoors, emulated a keyboard or mouse and exfiltrated data.

A hardware Trojan is usually composed of two parts: a trigger and payload. Triggers can activate payloads when a special condition is satisfied. The condition of a trigger is usually satisfied with very low-probability, so a payload can be activated with rare probability. When payload circuits are activated, malicious activities will occur. The aim of hardware Trojan detection is finding triggers and payloads. In different

* Corresponding author at: School of Automation, Northwestern Polytechnical University, Xian 710072, PR China.
E-mail addresses: 2016100110@mail.nwpu.edu.cn, shenlx@czu.cn (L. Shen), mudejun@nwpu.edu.cn (D. Mu), caog@czu.cn (G. Cao), qinmaoyuan@mail.nwpu.edu.cn (M. Qin), jblackst@ucsd.edu (J. Blackstone), kastner@ucsd.edu (R. Kastner).

https://doi.org/10.1016/j.cose.2018.07.006
0167-4048/© 2018 Elsevier Ltd. All rights reserved.

design objects, Trojans have different characteristics. A trigger may be classified as either an external trigger or an internal trigger. An internal-trigger uses an activation condition based on a particular input pattern, an internal logic state or counter value (Tehranipoor and Koushanfar, 2010). Time and data can be used as activation conditions of an internal-trigger. Hardware Trojans triggered with time are called time-bombs. Time-bombs cause serious threats to many high security areas because they are only affected by the internal system clock. System clocks do not need to be controlled by attackers who have to access to a hardware system. If a time-bomb is activated after a very long time, it will be very difficult to detect it because the testers may not have enough time to test all the code for time-bombs. Although formal validation techniques can verify all possible input values, it cannot prove that a time-bomb will never go off (Waksman and Sethumadhavan, 2011). Hardware Trojans triggered with data are called cheat codes. Cheat codes are the keys to identifying the payload of hardware Trojans. This work detects internal-trigger hardware Trojans.

Up to now, most literature focuses on post-fabrication detection which analyses IC chips, such as side channel technology. However, designers usually implement the functions at register-transfer level (RTL) code. Trojans inserted in the design at the register-transfer abstraction level or higher can be detected in RTL design. If Trojans are inserted in the gate-level netlist or later design stages, they can be detected by using equivalence checking tools for original RTL code (Fern and Cheng, 2016).

The aim of this study was to generate efficient test patterns for detecting internally triggered hardware Trojans in RTL code. We proposed a test generation method for hardware Trojans in RTL code. First, the synthesizable Verilog code was analyzed to generate the control flow graphs (CFGs). Next, parallel symbolic execution was implemented and a satisfiability modulo theories (SMT) solver was used to generate the test pattern. Finally, metamorphic testing detected internal triggered hardware Trojans by using the test pattern.

The contributions of our method include:

(1) We generated more precise and effective CFGs by representing a Verilog statement as a node of a CFG.
(2) We implemented a parallel symbolic execution algorithm for the synthesizable Verilog code. The algorithm generated test patterns which could detect hardware Trojans.
(3) Verilog expressions were converted to SMT-LIBv2 expressions. SMT-LIBv2 is supported by many popular SMT solvers, making it very flexible to choose an SMT solver to implement our method.
(4) Metamorphic testing was used to detect hardware Trojans. There is no need to use a golden circuit to compare the outputs of a Design Under Test (DUT) because metamorphic testing just verifies one or some metamorphic relationships among inputs and outputs.

The rest of the paper has the following structure: in Section 2 we discuss the hardware Trojan detection technologies in post-fabrication stage, physical design stage and functional design stage. In Section 3 we describe the details of our method which include the implementation of two phases: parallel symbolic execution based test generation and metamorphic testing. In Section 4 we analyze the Trust-Hub benchmarks by our method. In Section 5 we discuss the experimental results and conclude the features of our method as well as future work.

2. Background

2.1. *Detecting hardware Trojans in IC chips*

The detection difficulty is highest at this stage because of prohibitive time and cost requirements (Jacob et al., 2014). Current detection techniques mostly focus on post-fabrication stages and use a golden chip as a reference model. Side channel analysis and logic tests are two approaches in this stage (Bhunia et al., 2014). The side channel analysis can passively detect the hardware Trojan's side channel signal. Logic tests activate the hardware Trojans by using appropriate test patterns. Side channel analysis has been widely used to detect hardware Trojans because the inserted Trojans would affect the power consumption (Shende and Ambawade, 2016), current, signal delay and electromagnetic emanation (Ngo et al., 2015) of infected circuits. Unfortunately, side channel analysis requires long simulation time and relies on a golden model to compare the measured parameters for identifying a Trojan-inserted one. In many situations, it is difficult to obtain a golden model. Even if a golden model can be used, a small Trojan in a large circuit is very difficult to detect by side channel analysis because modern IC chips are becoming more and more complex.

2.2. *Detecting hardware Trojans in gate-level netlists*

The detection difficulty at the gate-level is medium (Jacob et al., 2014) because the netlist is used to produce IC chips. Compared with chips, gate-level netlists provide more design information. While many ATPG (Automatic Test Pattern Generation) tools are used at the gate-level, traditional ATPG tools are not useful for detecting hardware Trojans because their activation probability is very low. Random pattern test generation was proposed by Xue et al. (2014) dividing the DUT into regions based on heuristic partitions to reduce the analysis complexity. After this, a sequence of test vectors generated the ordering test vectors which introduced maximum switching activities in the regions. Finally, power ports were placed for localized transient current analysis. The generation of the test patterns was based on the circuit's structure and the power dissipation should be monitored during scan test. Some random algorithms have been used to generate the test patterns, such as the Genetic Algorithm. Saha et al. (2015) proposed a Genetic Algorithm based ATPG which was improved to detect small combinational and sequential hardware Trojans. The Genetic Algorithm detected many trigger conditions which were hard to be excited and the remaining unresolved trigger conditions were handled with a SAT(Boolean satisfiability) tool. The SAT tool returned the input vectors when trigger conditions were satisfiable.

Random methods are very time-consuming and cannot guarantee finding Trojans in limited amounts of time. Wang et al. (2016) proved this conclusion by some experiments and evaluated the signal probability to judge rare events which triggered small combinational Trojans. Test patterns were generated by comparing the probability with a threshold. The challenge of this method was how to find a compact set of patterns that cover all rare events.

Besides test pattern generation, some other methods have been proposed to detect hardware Trojans in netlists, such as information flow technology and pattern matching. Gate-level information flow tracking (Hu et al., 2014) realized the information flow technology in a gate-level circuit. By adding security labels to input signals, Gate-level information flow tracking generated a new gate-level circuit with security lattices. By tracking the information flow, new gate-level circuits are able to detect hardware Trojans. Although this approach is effective, adding labels to the original circuit increasing the complexity of the original netlist by 2^*n*. Here, "*n*" was the number of input signals.

2.3. *Detecting hardware Trojan in RTL design*

RTL code is synthesized by the tools to output the gate-level netlist. The gate-level circuit becomes very complex even with simple RTL code. Compared to the netlist, RTL code is more concise and easier to analyze. Jacob et al. (2014) thought that Trojan insertion in the RTL code was relatively easier than netlists and IC chips, and had a lower cost. The cost for Trojan detection in RTL code was lower than netlists and IC chips. It is very easy to insert hardware Trojans in RTL code, and it can be predicted that more and more RTL hardware Trojans will be designed in RTL code (Zhang and Xu, 2013). Our work focuses on Trojans in RTL code.

Traditional tools, such as ATPG, are not useful for RTL code because they are based on gate-level techniques (Mirzaei et al., 2013). To solve this, Banga and Hsiao (2010) proposed a Trojan detection technique in third party RTL using ATPG tools and equivalence checking. Mutation testing was used to detect hardware Trojans in Unspecified Functionality (Fern and Cheng, 2016). Mutation testing inserts artificial errors into the design code. If a mutation is detected, the test vector is useful. Otherwise, a new test vector should be generated. One drawback of mutation analysis is it's long run-time and the large manual effort required to analyze undetected mutants.

Unspecified hardware Trojans never violate the design specification because they do not alter the logic functions specified (Fern et al., 2017). To handle this issue, Fern used PyVerilog to directly analyze Verilog/VHDL code to detect unspecified hardware Trojans. SMT or boolean formulas for primary outputs were built from traversing the data-flow graph by PySMT. The formulas for "dangerous" functionality were transformed to satisfiability problems. If a formula was satisfied, the signal was flagged as dangerous. A mutual approach was used to identify (signal, condition) pairs which were the key to detecting unspecified hardware Trojans. PyVerilog (Takamaeda-Yamazaki, 2015) is a Hardware Design Processing Toolkit, which is written in Python for Verilog. This open-source toolkit consists of four libraries including a parser, data-flow analyzer, control-flow analyzer and Verilog code generator. Unlike PyVerilog, we implemented a control-flow analyzer by using Antlr4 and the outputs of our control-flow graph are different from PyVerilog. A node in our control-flow graph represents a statement instead of an operator or operand.

Some other methods which don't use test vectors have been proposed. Based on the principle of GLIFT, RTLIFT (Ardeshiricham et al., 2017) precisely measured all digital flows through RTL designs by adding Information Flow Tracking to the original Verilog code. The approach formally proved security properties related to integrity, confidentiality and logic side channels. For each operation, the number of output signals became twice of original input signals. The increasing number of input signals may increase the complexity and scale of original design circuit.

3. The symbolic execution based test pattern generation

As illustrated in Fig. 1, our method is composed of two phases.

Phase 1. The parse-tree of synthesizable Verilog code was generated by Antlr4 (Parr, 2013). Next, the CFGs of Verilog code were generated by our work. Based on the CFGs, parallel symbolic execution technology was implemented to get Path Conditions(PCs). PCs were solved with an SMT solver and the satisfiable test vectors were generated.

Phase 2. The satisfiable test vectors were analyzed with metamorphic testing to detect hardware Trojans.

In Phase 1, symbolic execution was used to discover the relationship between input variables and output variables. Obtaining the relationship is key to generating the test vectors which can discover Trojans. King (1976) proposed symbolic execution for program testing. Symbolic execution is a very useful program analysis technique. High-coverage test suites can be generated by symbolic execution and deep errors can be found too. It becomes practical because of the advances of constraint satisfiability. The basic idea is to represent the values of variables with the symbolic values of input variables. Symbolic execution can be implemented by the control flow of a program. Each execution path has a path condition which is a symbolic path constraint. During the process of symbolic execution, symbolic variables are mapped to symbolic expressions and Path Conditions(PCs). With the help of an SMT solver, PCs are solved. If a PC is satisfiable, an SMT solver can discover the values of input variables. A test vector set is generated according to the execution paths of the program. A symbolic execution path is composed of a sequence of conditional statements which are obtained from control flow graphs.

SMT can solve constraint-satisfaction problems. SMT is the core theory to solve the problems in many application areas, such as program analysis, test generation, verification. Modern SMT solvers decide the satisfiability of conjunctions of literals (De Moura and Bjørner, 2011). SMT provides a much richer modeling language than SAT. SMT-LIB calls an SMT solver that implements a procedure for satisfiability modulo theory (Barrett et al., 2017). SMT-LIB provides standard rigorous descriptions of background theories used in SMT systems, and it also develops and promotes common input and output languages for many different SMT solvers. Now SMT-LIBv2 is

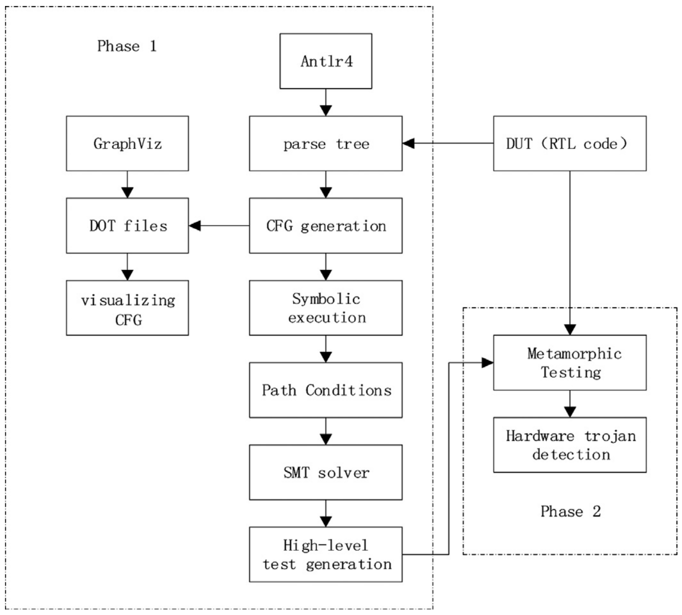

Fig. 1 – RTL test generation and Metamorphic testing hardware Trojan detection.

supported by many SMT solvers, such as Z3, Alt-Ergo, raSAT, SMTInterpol, SMT-RAT, Yices.

In Phase 2, the test generation was used to discover Trojans in RTL code. Usually, the outputs of test vectors are observed and used to compare the expectation results. In this study, we focused on metamorphic relationships rather than the value of output variables. Metamorphic testing (Chen et al., 1998) is used to check the correctness in software and alleviate the oracle problem. This method checks the relationships among inputs and outputs to discover abnormal code in programs. The relationships are called metamorphic relations. Metamorphic relations are the intrinsic properties of a program. A metamorphic relation violation refers to a checked error in a program (Yi et al., 2013). For example, let us consider a program that implements the cosine function. One of the properties of a cosine function is: $\cos(x) = \cos(-x)$. If there is the relation between inputs: $x_1 = x_2$, the relation of outputs should satisfy $\cos(x_1) = \cos(-x_2)$. If the relation of outputs does not satisfy $\cos(x_1) = \cos(-x_2)$, some faults must exist in the program. Metamorphic relations should be built according to the specifications of design.

3.1. *Test generation for Verilog code*

The control flow graph generated in this study was different from PyVerilog. Usually a node in a CFG represents an operator or a variable. In our work, a node in a CFG represented a statement in Verilog code.

3.1.1. *The node of a CFG*

Definition 1. a control flow graph is a directed graph, $CFG = <V, E>$. Where V is the set of vertices of the CFG and E is the set of edges of the CFG. A vertex $v, v \in V$, has the following characteristics:

(1) v is a quadruple(S, T, PRE, NEXT).
(2) S is a statement in Verilog code.
(3) T is the type of S. We define the types of synthesizable Verilog statements, such as ALWAYS and ALWAYS_END. An ENTER node is added to start the CFG, and an EXIT node is added to end the CFG.
(4) PRE is the set of previous nodes of S.
(5) NEXT is the set of next nodes of S.

3.1.2. *The edge of a CFG*

Definition 2. An edge $e = < v, u >, e \in E, v \in V, u \in V$ is the control relation between two nodes when Verilog code executes. An edge e has the following characteristics:

(1) The edge between two control nodes is a control relation.
(2) The edge between a control node and an assign node is a control relation.
(3) The edge between two assign nodes is an execution sequence, not a control relation.
(4) There is parallel execution among the statements of “always”, “instantiation” and “assign”.
(5) Nodes in PRE and NEXT can be found by control region of nodes and execution sequence.

3.1.3. *Control flow graph(CFG) generation for Verilog code*

In Algorithm 1, function CFGGeneration() read a Verilog file and extracted the basic grammar information from a parse tree. The most important grammar information for a node in a CFG includes *node.index*,*node.controlIndex*,*node.type*. *Node.index* is the index of a statement. *Node.controlIndex* is the index of control statement of the current node. The main grammar information of the lines 94–101 in the Appendix B was listed in Table 1. The control relationship is described in Fig 2. The function connectControlFlowListVariable() created the edge of CFG.

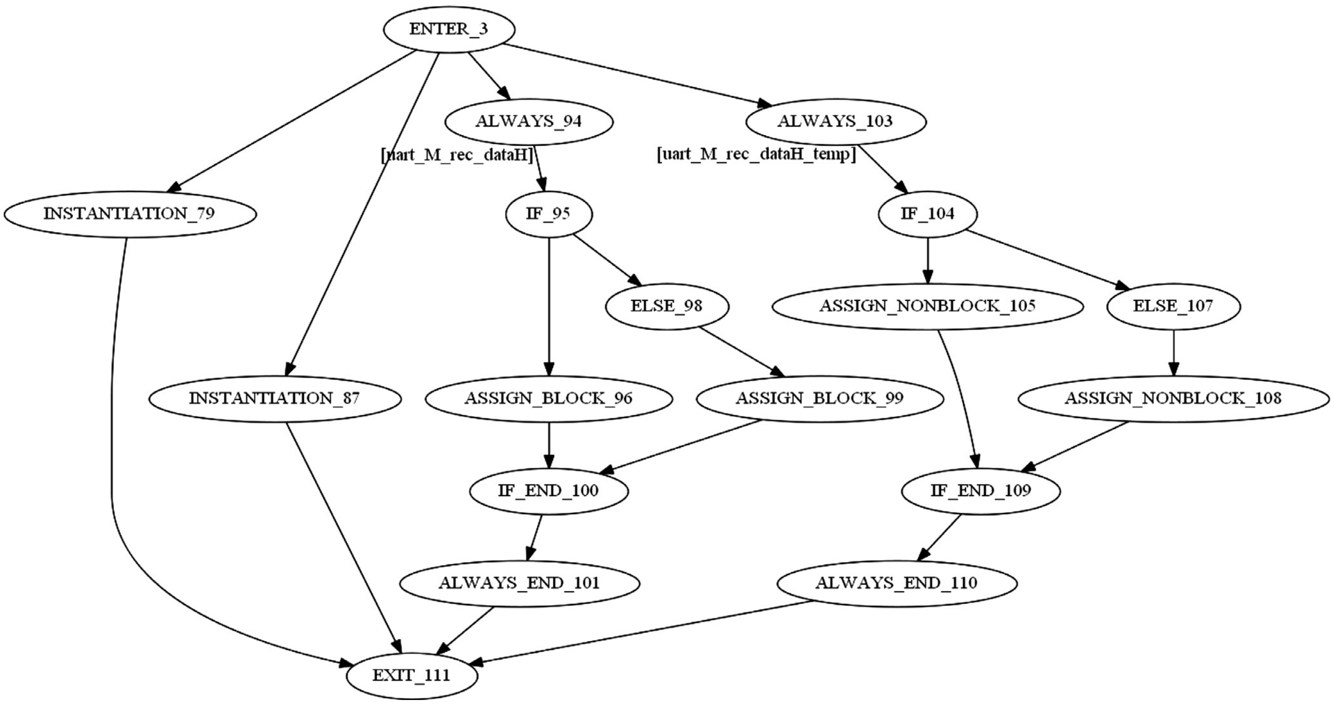

Fig. 2 – The CFG of uart.v (drawn by Graphviz).

3.1.4. *Parallel symbolic execution*

The symbolic execution of Verilog code is described in Algorithm 2. Algorithm 2 is a recursive function. The function fileControlFlowList() is the start of the entire analysis. The input of the algorithm is a synthesizable Verilog file(VC.v). The CFGs of VC.v were generated by Algorithm 1. Next, the filename of instantiation statements were obtained and the instantiation nodes were analyzed by calling fileControlFlowList(). After the code of instantiation files(subModuleVC.v) were analyzed, the PCs(subPCs) of non-input variables were returned. When all the subPCs were returned, the parallel symbolic execution thread symbolExecutionThread() started to analyze the always statements and continuous assign statements. The threads returned the PCs of variables at the left of statements.

A flag variable, *nonInput_variable.isDone*, would be set "true" if the PCs of a non-input variable were obtained. *Input_variable.isDone* would be set "true" in the stage of initialization. When *nonInput_variable.isDone* was set "true", the sub-threads which were halted because the *nonInput_variable.isDone* was equal to "false" would continue to run.

An optimization during the progress of parallel symbolic execution is possible. If PCs are not satisfiable, they may be deleted during symbolic execution, reducing the number of PCs and consequently the execution time and memory required.

3.1.5. *Loop dependency problem and randomizing internal variables*

Loop dependencies are a complex situation in RTL code. For example, the code in Listing 1 has loop dependencies because the variable "count_in" exists at both sides of "$<=$" at the same time. "count_in" is a non-input variable.

During symbolic execution, we replaced the variable on the left side of an assignment with the variable on the right side. If a variable is on both sides of an assignment,

Table 1 – The main grammar information of Verilog code.

	Node	Node.index	Node.controlIndex	Node.type
94	always @(posedge sys_clk or negedge sys_rst_l) begin	3	0(ENTER)	ALWAYS
95	if (~sys_rst_l) begin	4	3	IF
96	rec_dataH = 0;	5	4	ASSIGN_BLOCK
97	end			
98	else begin	6	4	ELSE
99	rec_dataH = rec_dataH_temp;	7	6	ASSIGN_BLOCK
100	end	8	4	IF_END
101	end	9	3	ALWAYS_END

Algorithm 1 – CFG generation.

```
Input: synthesizable verilog code(filename.v)
Output: CFG
CFG CFGGeneration(filename.v)
{
  ENTER→CFG; // add ENTER node into CFG
  Walk parse tree of filename.v and create v, v→CFG;
  EXIT→CFG; // add EXIT node into CFG
  Get v from CFG;
  while (! (all nodes are handled))
  {
    u← CFG; //get u from CFG;
    //Adjust the PRE and NEXT of v and u
    connectControlFlowListVariable();
    v← u; // the value of u overwrites the value of v
  }
  Breadth-first traverses CFG to generate the DOT files of CFG;
  Draw CFG with DOT files;
  return CFG;
}
```

Algorithm 2 – parallel symbolic execution.

```
Input: synthesizable verilog code(VC.v)
Output: the set of PC
PC fileControlFlowList(VC.v)
{
  CFG←CFGGeneration(VC.v);
  for (each INSTANTIATION node)
  {
    subPC = fileControlFlowList(subModuleVC.v);
    noninput_variables.PC← subPC;
  }
  return noninput_variables.PC← symbolExecution(CFG);
}
PC symbolExecution(CFG)
{
  initialize noninput_variables.PC;
  input_variable.isDone = true;
  nonInput_variable.isDone = false;
  for (each ALWAYS or ASSIGN_CONTINUOUS node)
    //Parallel execute sub-thread
    noninput_variables.PC← symbolExecutionThread(node_index,
    CFG); return noninput_variables.PC;
}
PC symbolExecutionThread(node_index, CFG)
{
  root ←node_index;
  Depth-first traverses CFG, start from root;
  while (Depth-first traverse not end)
  {
    get a node from CFG; // assign statement will be handled
    if (node is an assign statement)
    {
      for (each variable in the right of statement)
      {
        while (variable.isDone == false)
          hang-up current symbol execution thread;
      }
    }
    PC← replace noninput_variables at the right of statement with
      input_variables;
    if (PC is satisfiable)   update left_variables.PC;
  }
  left_variables.isDone = true;
  return left_variables.PC;
}
```

it means the variable is replaced by itself. The result cannot be solved by symbolic execution which uses input variables and constants to express an output variable. As an internal counter, “count_in” is only affected by the system clock. The system clock is a special signal which cannot be directly added to the expression of “count_in” because the system clock variable “sys_clk” never appears in the right side of an assignment. In our experimental Verilog code there are two assignments about “count_in”: “count_in <= 32′h0” and “count_in <= count_in + 1′b1”. If “count_in” is replaced with “0”, the symbolic expression can be solved during symbolic execution. Replacing “count_in” with “count_in + 1” leads to an endless loop and symbolic execution cannot go on because “count_in” isn’t an input variable or constant.

To solve the problem of loop dependencies, during symbolic execution we converted “count_in” into a random variable which was regarded as an input variable causing some internal variables to become global random variables. The global random variables revealed some important internal conditions. For the code in Listing 1, “count_in” was replaced with “RANDOM_filename_M_count_in”. The path condition “count_in == 32′hffffffff” equalled “RANDOM_filename_M_count_in == 32′hffffffff”. The solver got a satisfiable result: “RANDOM_filename_M_count_in = #xffffffff”. One PC for “count_in” was “RANDOM_filename_M_count_in == 0xffffffff” and the other PC for “count_in” was “RANDOM_filename_M_count_in! = 0xffffffff”. “count_in” is an internal variable, the trigger of a hardware Trojan, and a counter which determines the value of the system clock. When “count_in” was equal to “0xffffffff”, a hardware Trojan payload was activated.

Converting internal variables into random variables can overapproximate the set of values that the variable takes during the course of execution, which will lead to spurious states and false positives. To solve this problem, we used constraint conditions to represent the internal variables’ special values range. For example, we represented “count_in >= 5 and count_in <= 0xfffffffe” with a constraint condition and we added the constraint condition to the path conditions after symbolic execution. A new path condition has the SMT-LIBv2 format: (and (constraint conditions) (a path condition)). If a new path condition is not solved, a satisfiable result cannot be achieved. As a result, the overapproximation problem can be avoided. To prove the function of constraint conditions, the Verilog code in Listing 1 was modified in Listing 2.

Listing 1 – A loop dependency in Verilog code.

```
if (count_in == 32′hffffffff) begin
  DataSend_ena <= 1′b1;
  count_in <= 32′b0;
end
else
  count_in <= count_in + 1′b1;
```

Listing 2 – Verilog code with infeasible trigger.

```
if (~sys_rst_l) begin
  DataSend_ena <= 1'b0;
  count_in <= 32'h5; // count_in's values from 5 to 0xfffffffe
end
else if (count_in == 32'hfffffffe) begin
    DataSend_ena <= 1'b1;
    count_in <= 32'h5; // count_in's values from 5 to 0xfffffffe
  end
  else if (count_in < 5) // infeasible condition
          count_in <= 32'hfffffffe;
       else  count_in <= count_in + 1'b1;
```

Listing 3 – The SMT-LIBv2 statements include the infeasible condition. They can be tested in Z3 (https://rise4fun.com/Z3/tutorial/guide). The result is "unsat" (unsatisfiable).

```
(declare-const RANDOM_u_xmit_M_count_in (_ BitVec 32))
(assert (and (and (bvule RANDOM_u_xmit_M_count_in #xfffffffe)
(bvuge RANDOM_u_xmit_M_count_in #x00000005)) (bvult
RANDOM_u_xmit_M_count_in #x00000005)))
(check-sat)
```

In Listing 2 the constraint condition of "count_in" is "(and (bvule count_in #xfffffffe) (bvuge count_in 5))". The path condition of "count_in < 5" is "(bvult count_in 5)". The total path condition is

"(and (and (bvule count_in #xfffffffe)
(bvuge count_in 5))
(bvult count_in 5))".

During symbolic execution, we replaced "count_in" with "RANDOM_filename_M_count_in ", the condition was transformed to:

"(and (and (bvule RANDOM_filename_M_count_in #xffffffffe)
(bvuge RANDOM_filename_M_count_in 5))
(bvult RANDOM_filename_M_count_in 5))"

Listing 3 consists of the SMT-LIBv2 statements used to test the infeasible condition. The result is unsatisfiable, so no according test vector is generated and metamorphic testing does not handle the infeasible condition.

3.1.6. SMT solver and test generation

PCs which had the Verilog grammar format were converted to SMT-LIBv2 format in our method. Afterwards, Z3 (Microsoft, 2017) which was a SMT solver from Microsoft Research was used to solve the PCs.

Convert Verilog statement to SMT-LIBv2 format: A Verilog statement is an infix expression while a SMT-LIBv2 statement is prefix expression. Z3 cannot solve a PC which is an infix expression. An infix expression should be converted to a prefix expression. To solve this problem, Algorithm 3 was proposed. During the conversion, Verilog operators were replaced with SMT-LIBv2 operators. The operator conversion between Verilog and SMT-LIBv2 are listed in Appendix A.

The function convertToBitVector(*element*) changed the format of "*element*" to SMT-LIBv2. The function convertToSMT-

Algorithm 3 – Convert infix expression of Verilog to prefix expression of SMT-LIBv2.

```
Input: infix expression(middle) of Verilog
Output: prefix expression(prefix) of SMT-LIBv2
prefix middleToPrefix(middle)
{
  tmp ← middle;
  for (each element1 in tmp)
  {
    tmpPrefix ← element1;
    if (element1.type == OPERATOR)
      operatorStack ← element1;
  }
  for (each element2 in tmpPrefix)
  {
    if (element2.type == CONST)
      dataStack ← convertToBitVector(element2);
    if (element2.type == VARIABLE)
      dataStack ← element2;
    if (element2.type == OPERATOR)
    {
      operatorStack ← convertToSMTOperator(element2);
      tmpPrefix ← operatorHandle(dataStack,operatorStack);
    }
  }
  return tmpPrefix;
}
```

Operator(*operator*) converted the "*operator*" of Verilog to SMT-LIBv2 operator. The function operatorHandle() dealt with the priority of operators.

For example, an infix expression of Verilog,

(rec_dataH_rec == xmit_dataH) && (rec_dataH_rec == {x_START,x_WAIT,x_SHIFT[1:0]}), was converted to prefix expression SMT-LIBv2:

(add (= rec_dataH_rec xmit_dataH)
(= rec_dataH_rec
(concat x_START (concat x_WAIT
((_ extract 1 0) x_SHIFT))))).

Test generation: If a PC was satisfiable, the values of input variables were obtained. For example, after Z3 solved the statement:

(rec_dataH_rec == xmit_dataH) && (rec_dataH_rec == 76), the result from the Z3 was "rec_dataH_rec = xmit_dataH and rec_dataH_rec = 76". We got an input vector which included "xmit_dataH = 76"("rec_dataH_rec" is not an input variable).

3.2. Metamorphic testing

To verify the test generation generated in Section 3.1, metamorphic testing was used to detect the hardware Trojan. In our research, Trust-Hub RS-232 benchmarks were used, so we defined the metamorphic relation of RS-232 as following:

$$(X_i = X_j) \Rightarrow (Y_i \oplus Y_j = 0) \tag{1}$$

where X is an 8-bits data input which should be sent out by a sender. Y is an 8-bit data output which is acquired by the receiver. The function of RS-232 is that a sender transmits X

Table 2 – The Trojan description of T300, T400, T500 (Shakya et al., 2017; Salmani et al., 2013).

	T300	T400	T500
Trojan trigger	Trigger is a 32-bit counter("count_in"). When "count_in" reaches 32′hFFFFFFFF, the payload becomes activated.	Trigger compares transmitted and received data("xmit_dataH", "rec_dataH_rec"). If both equal 8′h4C the payload becomes activated.	Trigger is a 32-bit counter("count_in"). When "count_in" reaches 32′hFFFFFFFF the payload becomes activated.
Trojan payload	Payload replaces the 7th bit of all transmitted data after the payload was activated.	4 bits of received data are replaced by the payload.	"xmit_doneH" is stuck at '0'.
Insertion phase	design	design	design
Abstraction level	Register-transfer level (RTL)	Register-transfer level (RTL)	Register-transfer level (RTL)
Activation mechanism	Internally time-based triggered	Internally conditionally(data-based) triggered	Internally time-based triggered

to a receiver who gets X as Y. If X is received correctly, Y should be equal to X. According to formula (1), let $X_i = X_j$, then
if $(Y_i \oplus Y_j = 0)$, Y_i and Y_j may be normal,
if $(Y_i \oplus Y_j \neq 0)$, Y_i and Y_j are abnormal.

Because "$0 \oplus 1 = 1$" and "$0 \oplus 0 = 0$", "0" can detect abnormal "1" by XOR. Because "$1 \oplus 0 = 1$" and "$1 \oplus 1 = 0$", "1" can detect abnormal "0" by XOR. So when 8-bits of "0"(00000000) are sent to a receiver, "00000000" should be received. If Y is not equal to "00000000", abnormal "1"s and their locations can be detected according to the result of $(Y_i \oplus Y_j)$. If 8-bits of "1"(11111111) are sent, abnormal "0"s and their locations can be detected in the same way.

For example, "$X = (0000,0000)_2$" was sent twice, then "$Y_i = (0000,0000)_2$" and "$Y_j = (0100,0111)_2$" were received. "$Y_i \oplus Y_j = (0100,0111)_2$ " meant that four abnormal "1"s were detected at 6th,2th,1th,0th bit.

In-fact, it is very important to test the metamorphic relation in many network protocols.

4. Experiments and results

The time-bomb and cheat codes in RTL codes are difficult to detect. To detect them, symbolic execution and metamorphic testing were used in this study. The Trust-Hub benchmark (Salmani et al., 2013; Shakya et al., 2017) RS-232-T300, T400 and T500 were analyzed. Compared with PyVerilog, our work generated more precise CFGs and disclosed the relationships between statements. By using an SMT solver, test patterns were compacted effectively which increased the speed of analysis. The Trojans in the three benchmarks were detected by the test vectors and the abnormal bits of output variables were detected.

Three Verilog files were analyzed: uart.v, u_xmit.v, u_rec.v. The source code of uart.v in T300 was attached in Appendix B. Table 2 describes the details of three Trojans. T300 and T500 had time triggers. T400 had a data trigger.

4.1. *Test generation for Verilog code*

4.1.1. *Control flow graph generation for Verilog code*

To identify the variables in all Verilog files, we added a prefix before each variable. The prefix has the format: "file-name_M_". So the variables in uart.v would add a prefix "uart_M_". If a variable is changed to a random variable, a prefix "RANDOM_" will be added.

Table 3 – The number of test vectors of each output variable.

Output variable	The number of test vectors
uart_M_rec_dataH	3
uart_M_uart_XMIT_dataH	6
uart_M_xmit_doneH	4
uart_M_rec_readyH	4

Before symbolic execution, the CFGs of RTL code were generated. The CFG of "uart.v" in Trust-Hub RS-232 T300 is shown in Fig. 2. The name of node has the format: "node-type_lineNo". So "INSTANTIATION_79" in Fig. 2 means the node type is INSTANTIATION and the instantiation statement is at line 79 in uart.v. "[uart_M_rec_dataH]" meant that the variable "rec_dataH" was changed between always statements line 94 and 101. ENTER is the first node and EXIT is the last node.

4.1.2. *Symbolic execution and test generation*

The test vectors were generated for each output variable. It was convenient to test one output signals or all output signals.

Trust-Hub RS-232 T300: Table 3 shows the results of symbolic execution. In fact, some of the test vectors are the same for different output variables. For instance, reset signal "sys_rst_l" resets the whole circuit and affects all outputs. Table 4 shows the detail of test vectors of "uart_M_uart_XMIT_dataH". The first test vector means that the reset signal "uart_M_sys_rst_l" (in top file) is enabled. The second test vector means that an input signal "xmitH" in uart.v was "0" and the reset signal "uart_M_sys_rst_l" is disabled. But two internal signals "RANDOM_u_xmit_M_count_in" and "RANDOM_u_xmit_M_bitCell_cntrH" must satisfy conditions "0xFFFFFFFF" and "15", respectively. When "count_in" was equal to "0xFFFFFFFF", it activated a Trojan payload by setting internal signal "DataSend_ena = 1". Detection detail was described in Section 4.2. "uart_M_xmit_dataH(random)" means

Table 4 – The test vectors for "uart_M_uart_XMIT_dataH".

	Input Vectors	Explanation
1	uart_M_sys_rst_l = 0	Reset

2	uart_M_sys_rst_l = 1
	uart_M_xmitH = 0
	RANDOM_u_xmit_M_count_in = 0xFFFFFFFF
	RANDOM_u_xmit_M_bitCell_cntrH = 15
	uart_M_xmit_dataH(random)
		Activate the hardware Trojan

3	uart_M_sys_rst_l = 1
	uart_M_xmitH = 1
	RANDOM_u_xmit_M_count_in = 0xFFFFFFFF
	RANDOM_u_xmit_M_state = 0
	uart_M_xmit_dataH(random)	

4	uart_M_sys_rst_l = 1
	uart_M_xmitH = 0
	RANDOM_u_xmit_M_bitCell_cntrH = 15
	uart_M_xmit_dataH(random)
		Normal output

5	uart_M_sys_rst_l = 1
	uart_M_xmitH = 1
	RANDOM_u_xmit_M_bitCell_cntrH = 15
	uart_M_xmit_dataH(random)	

6	uart_M_sys_rst_l = 1
	uart_M_xmitH = 1
	RANDOM_u_xmit_M_bitCell_cntrH = 1
	uart_M_xmit_dataH(random)	

Listing 4 – Trigger circuit in T300 (Shakya et al., 2017;Salmani et al., 2013).

```
always @ (negedge sys_rst_l or posedge xmitH) begin
  if (~sys_rst_l) begin
    DataSend_ena < = 1'b0;
    count_in < = 32'h0;
  end else if (count_in = = 32'hffffffff) begin
              DataSend_ena < = 1'b1;//trigger
              count_in < = 32'h0;
        end else
                  count_in < = count_in + 1'b1;
end
```

Listing 5 – Trigger circuit in T400 (Shakya et al., 2017; Salmani et al., 2013).

```
always @(posedge xmit_doneH or negedge sys_rst_l) begin
  if (~sys_rst_l) begin
    cntr < = 1'b0;
  end
  else begin
      if((rec_dataH_rec = = xmit_dataH) && (rec_dataH_rec = =
      {x_START, x_WAIT, x_SHIFT[1:0]})) // trigger
        cntr < = 1'b1;
      else
        cntr < = 1'b0;
end
```

that the input variable "uart_M_xmit_dataH" could be any value in its value ranges. "RANDOM_u_xmit_M_state = 0" means the start state of a finite-state machine in u_xmit.v. Listing 4 shows the source code of trigger. We detected this internal trigger condition by randomizing internal variable "count_in" in u_xmit.v during the process of symbolic execution.

Trust-Hub RS-232 T400: Listing 5 shows the code of the trigger in T400. The trigger's condition is "(rec_dataH_rec = = xmit_dataH) && (rec_dataH_rec = = {x_START, x_WAIT, x_SHIFT[1:0]})". The condition means that input 8-bits "xmit_dataH" was equal to output 8-bits "rec_dataH" and both of them were equal to "76"({x_START, x_WAIT, x_SHIFT[1:0]} = 010 011 00). Table 5 lists the number of test vectors of each output variable. The second test vector in Table 6 activates the payload. Detection detail was described in Section 4.2.

Trust-Hub RS-232 T500: Listing 6 shows the trigger circuit in T500. Table 7 lists the number of test vectors of each output variable. Table 8 lists the test vectors of "uart_XMIT_dataH"

Table 5 – The number of test vectors of each output variable.

Output variable	The number of test vectors
uart_M_rec_dataH	3
uart_M_uart_XMIT_dataH	6
uart_M_xmit_doneH	5
uart_M_rec_readyH	5

Table 6 – The test vectors for "uart_M_rec_dataH".

	Input vectors	Explanation
1	uart_M_sys_rst_l = 0	Reset

2	uart_M_sys_rst_l = 1
	uart_M_xmit_dataH = 76
	uart_M_uart_REC_dataH = 0
		Activate the hardware Trojan

Others		Normal output

Listing 6 – Trigger circuit in T500 (Shakya et al., 2017; Salmani et al., 2013).

```
always @ (negedge sys_rst_l or posedge sys_clk) begin
   if (~sys_rst_l) begin
      DataSend_ena <= 1'b0;
      count_in <= 32'h0;
   end else if (count_in == 32'hffffffff) // trigger
                DataSend_ena <= 1'b1;
           else
                count_in <= count_in + 1'b1;
end
```

Table 7 – The number of test vectors of each output variable.

Output variable	The number of test vectors
uart_M_rec_dataH	3
uart_M_uart_XMIT_dataH	5
uart_M_xmit_doneH	3
uart_M_rec_readyH	3

in uart.v. The Trojan was activated when the second and third test vectors were used. Detection detail was described in Section 4.2.

4.2. Metamorphic testing

According to Section 3.2, let $X = \text{uart\_M\_xmit\_dataH}$, $Y = \text{uart\_M\_rec\_dataH}$. Input X was used twice. The first X was X_i, and the second X was X_j, making X_i equal to X_j. The test vectors in Section 4.1 were used after X_i was set and before X_j was set. The abnormal outputs were detected after a test vector was used. The object of the experiments was to test the metamorphic relationship between Y_i and Y_j. Hardware Trojans were detected according to the results of metamorphic testing.

4.2.1. Trust-Hub RS-232 T300

In Table 9, Y_i is the output of X_i before the second and third test vectors in Table 4 were used and Y_j is the output of X_j after the two test vectors were used. According to the results of ($Y_i \oplus Y_j$) in Table 9 and 8 abnormal outputs were detected. By using “0”, one abnormal “1” was detected at the 7th bit in the result of ($Y_i \oplus Y_j$). Figs. 3 and 4 show the result affected by the Trojan after the test vectors were used. In Fig. 3 the internal variable “count_in” in u_xmit.v reaches “0xFFFFFFFF” at the location where X was equal to “0 x00” again. Before X_j was set “0 x00”, $X_i = Y_i$. After X_j was set “0 x00”, some abnormal Y_j were detected in Fig. 3. In Fig. 4, X was set “0xff” and one abnormal bit is detected. The data in Fig. 3 is shown in the left part of Table 9 and the data in Fig. 4 is shown in the right part. The hardware Trojan was detected by the randomized internal variable. The location of abnormal bits was consistent with the Trojan description in Table 2. The branch coverage of uart.v, u_xmit and u_rec.v were 100%, 100% and 93.8%, respectively.

4.2.2. Trust-Hub RS-232 T400

There is a data trigger in T400. After X_j was set “0 x4c”, two abnormal outputs are detected in the result of ($Y_i \oplus Y_j$) in Table 10. By using “0”, two abnormal “1”s are detected at the 0th and 6th bits in ($Y_i \oplus Y_j$). By using “1”, two abnormal “0”s are detected at the 5th and 7th bit. Figs. 5 and 6 were the results generated by simulating in QuestaSim 10. The data in Fig. 5 is listed in the left part of Table 10 and data in Fig. 6 is listed in the right part. The hardware Trojan was detected by the special input value “0 x4c”. The locations of abnormal bits were in line with the Trojan description in Table 2. The branch coverage of uart.v, u_xmit and u_rec.v were 100%, 100% and 93.8%, respectively.

4.2.3. Trust-Hub RS-232 T500

The value of internal variable “count_in” in u_xmit.v reaches “0xFFFFFFFF” before the X was set “0 x00” or “0xff” again. After “count_in” reached “0xFFFFFFFF” some abnormal outputs were detected in Figs. 7 and 8. By using “0”, eight “1”**s** are detected in ($Y_i \oplus Y_j$), so all bits are abnormal. By using “1”, eight “1”**s** are detected, so all bits are abnormal. Figs. 7 and 8 are the results by simulating in QuestaSim 10. The abnormal “xmit_done” is found in Figs. 7 and 8 when abnormal results were detected in ($Y_i \oplus Y_j$) in Table 11. The hardware Trojan was detected by the randomized internal variable. The abnormal outputs(“rec_dataH” and “xmit_doneH”) were consistent with the Trojan description in Table 2. The branch coverage of uart.v, u_xmit and u_rec.v were 100%, 97.6% and 93.8%, respectively.

Table 8 – The test vectors for “uart_M_uart_XMIT_dataH”.

	Input vectors	Explanation
1	uart_M_sys_rst_l = 0	Reset

2	uart_M_sys_rst_l = 1
	uart_M_xmitH = 0
	RANDOM_u_xmit_M_count_in = 0xFFFFFFFF
	uart_M_xmit_dataH(random)
		Activate the hardware Trojan

3	uart_M_sys_rst_l = 1
	uart_M_xmitH = 1
	RANDOM_u_xmit_M_count_in = 0xFFFFFFFF
	uart_M_xmit_dataH(random)	

Others		Normal

Table 9 – The results generated by QuestaSim 10.1b. Y_i are the output of Trust-Hub RS232-T300 before Input vector was set the value in the second and third test vectors in Table 4. Y_j is the output after the two test vectors were set.

Input X	Output Y_i	Output Y_j	$(Y_i \oplus Y_j)$	Input X	Output Y_i	Output Y_j	$(Y_i \oplus Y_j)$
0x0	0x0	0 x80	1000,0000	0 xff	0 xff	0 xff	0000,0000
0 x1	0 x1	0 x81	1000,0000	0xfe	0 xfe	0xfe	0000,0000
0 x2	0 x2	0 x82	1000,0000	0xfd	0xfd	0xfd	0000,0000
0 x4	0 x4	0 x84	1000,0000	0xfb	0xfb	0xfb	0000,0000
0 x8	0 x8	0 x88	1000,0000	0xf7	0xf7	0xf7	0000,0000
0 x10	0 x10	0 x90	1000,0000	0xef	0xef	0xef	0000,0000
0 x20	0 x20	0xa0	1000,0000	0xdf	0xdf	0xdf	0000,0000
0 x40	0 x40	0xc0	1000,0000	0xbf	0xbf	0xbf	0000,0000
0 x80	0 x80	0 x80	0000,0000	0 x7f	0 x7f	0xff	1000,0000

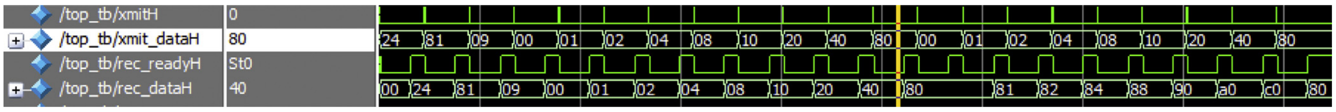

Fig. 3 – T300-1.

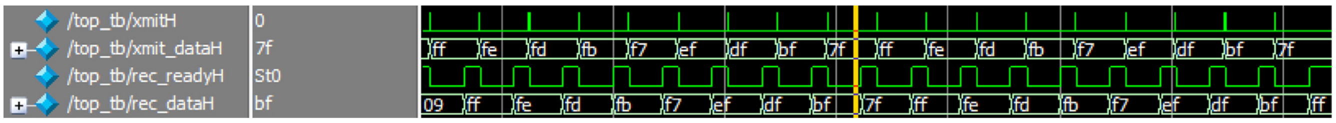
/top_tb/xmitH 0
/top_tb/xmit_dataH 7f
/top_tb/rec_readyH St0
/top_tb/rec_dataH bf
ff fe fd fb f7 ef df bf 7f ff fe fd fb f7 ef df bf 7f
09 ff fe fd fb f7 ef df bf 7f ff fe fd fb f7 ef df bf ff

Fig. 4 – T300-2.

Table 10 – The results generated by QuestaSim 10.1b. The second test vector in Table 6 was used.

Input X	Output Y_i	Output Y_j	$(Y_i \oplus Y_j)$	Input X	Output Y_i	Output Y_j	$(Y_i \oplus Y_j)$
0 x4c	0 x4c	0 x4c	0	0 x4c	0 x4c	0 x4c	0
0 x0	0 x0	0 x41	0100,0001	0xff	0xff	0 x5f	1010,0000
0 x1	0 x1	0 x1	0	0xfe	0xfe	0xfe	0
0 x2	0 x2	0 x2	0	0xfd	0xfd	0xfd	0
0 x4	0 x4	0 x4	0	0xfb	0xfb	0xfb	0
0 x8	0 x8	0 x8	0	0xf7	0xf7	0xf7	0
0 x10	0 x10	0 x10	0	0xef	0xef	0xef	0
0 x20	0 x20	0 x20	0	0xdf	0xdf	0xdf	0
0 x40	0 x40	0 x40	0	0xbf	0xbf	0xbf	0
0 x80	0 x80	0 x80	0	0 x7f	0 x7f	0 x7f	0

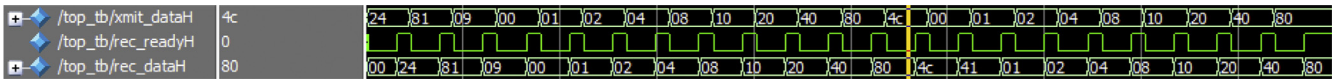
/top_tb/xmit_dataH 4c
/top_tb/rec_readyH 0
/top_tb/rec_dataH 80
24 81 09 00 01 02 04 08 10 20 40 80 4c 00 01 02 04 08 10 20 40 80
00 24 81 09 00 01 02 04 08 10 20 40 80 4c 41 01 02 04 08 10 20 40 80

Fig. 5 – T400-1.

5. Discussions and conclusions

To generate test patterns for hardware Trojan detection, control flow analysis, symbolic execution, SMT and metamorphic testing were used in our work. CFGs were generated by analyzing the grammar of the Verilog code. By walking the CFGs, Symbolic execution executed the multi-threads to replace the non-input variables with input variables and random variables in PCs. Z3 was used to solve the PCs to discover the satisfiable input vectors. The satisfiable input vectors were detected by metamorphic testing to detect the abnormal outputs which may be caused by hardware Trojans.

In our work, the key was the randomizing of the internal variables during symbolic execution. The randomizing found out the internal variables which were affected by internal conditions. Path Conditions including randomizing internal variables were also constructed, which detected the internal

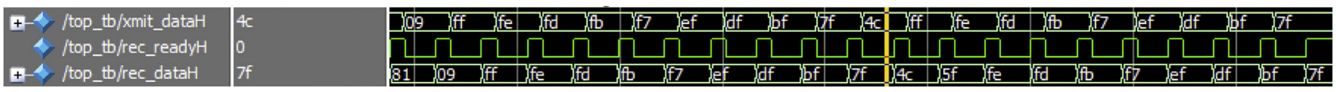

Fig. 6 – T400-2.

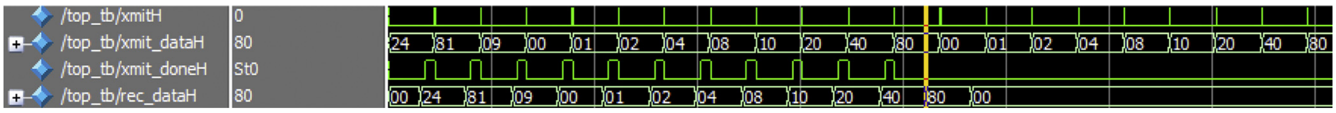

Fig. 7 – T500-1.

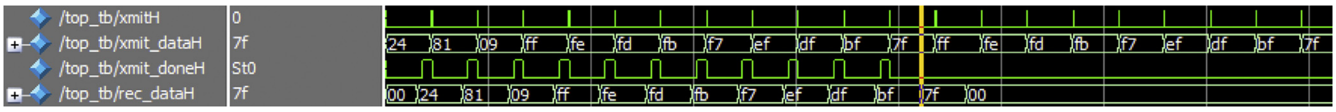

Fig. 8 – T500-2.

Table 11 – The results generated by QuestaSim 10.1b. The second and third test vectors in Table 8 were used.

Input X	Output Y_i	Output Y_j	$(Y_i \oplus Y_j)$	Input X	Output Y_i	Output Y_j	$(Y_i \oplus Y_j)$
0x0	0x0	0	0000,0000	0xff	0xff	0	1111,1111
0x1	0x1	0	0000,0001	0xfe	0xfe	0	1111,1110
0x2	0x2	0	0000,0010	0xfd	0xfd	0	1111,1101
0x4	0x4	0	0000,0100	0xfb	0xfb	0	1111,1011
0x8	0x8	0	0000,1000	0xf7	0xf7	0	1111,0111
0x10	0x10	0	0001,0000	0xef	0xef	0	1110,1111
0x20	0x20	0	0010,0000	0xdf	0xdf	0	1101,1111
0x40	0x40	0	0100,0000	0xbf	0xbf	0	1011,1111
0x80	0x80	0	1000,0000	0x7f	0x7f	0	0111,1111

conditions triggered only by system clock or a special internal constant. Unlike the random pattern test generation, our work detects the triggers with more accuracy and certainty, less randomness. Because the test vectors are generated according to statements branch of RTL code, our method has very high branch coverage.

Our work can be used during the process of the RTL design to discover suspicious conditions and internal variables. It also provides a more accurate and concise test generation which can also be used to detect the design errors in RTL code written by Verilog DHL. This work is a very important base for future effort to detect the more complex internal time and data triggers.

Acknowledgments

This work was supported by the National Cryptography Development Fund [grant number MMJJ20170210]; the National Natural Science Foundation of China [grant number 61672433] and the Fundamental Research Fund of Shenzhen Science and Technology Innovation Committee under Grants 201703063000511 and 201703063000517, the Social Science Foundation of Jiangsu Province[grant number 16GLB014].

Appendix A: The correspondence between Verilog operators and SMT-LIBv2 operators

Verilog operator	SMT-LIBv2 operator
!, ~	bvnot
*	bvmul
/	bvudiv
%	bvmod
+	bvadd
−	bvsub
<<	bvshl
>>	bvlshr, bvashr
>	bvugt, bvsgt
>=	bvuge, bvsge
<	bvult, bvslt
<=	bvule, bvsle
==	=
&&	and
&	bvand
|	bvor
^	bvxor
^~	bvxnor
||,or	or
,(in {})	concat

Appendix B: The source code of uart.v in RS-232-T300

```
u_xmit iXMIT(.sys_clk(sys_clk),
.sys_rst_l(sys_rst_l),
.uart_xmitH(uart_XMIT_dataH),
.xmitH(xmitH),
.xmit_dataH(xmit_dataH),
.xmit_doneH(xmit_doneH)
);

u_rec iRECEIVER (.sys_rst_l(sys_rst_l),
  .sys_clk(sys_clk),
  .uart_dataH(uart_REC_dataH),
  .rec_dataH(rec_dataH_rec),
  .rec_readyH(rec_readyH)
  );

always @(posedge sys_clk or negedge sys_rst_l) begin
  if (~sys_rst_l) begin
    rec_dataH = 0;
end
  else begin
    rec_dataH = rec_dataH_temp;
  end
end

always @(posedge rec_readyH or negedge sys_rst_l) begin
  if (~sys_rst_l) begin
    rec_dataH_temp <= 0;
end
else begin
      rec_dataH_temp <= rec_dataH_rec;
  end
end
endmodule
```

REFERENCES

Ardeshiricham A, Hu W, Marxen J, Kastner R. Register transfer level information flow tracking for provably secure hardware design. Proceedings of conference on design, automation & test in Europe conference & exhibition (DATE); 2017. p. 1695–700.

Banga M, Hsiao M. Trusted RTL_ Trojan detection methodology in pre-silicon designs. Proceedings of IEEE international symposium on hardware-oriented security & trust; 2010. p. 56–9.

Barrett C, Fontaine P, Tinell C. The SMT-LIB standard v2.6. 2017.

Bhunia S, Hsiao MS, Banga M, Narasimhan S. Hardware Trojan attacks: threat analysis and countermeasures. Proc IEEE 2014;102(8):1229–47.

Chen TY, Cheung S, Yiu SM. Metamorphic testing: a new approach for generating next test cases, Technical Report HKUST-CS98-01. Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong; 1998.

De Moura L, Bjørner N. Satisfiability modulo theories. Commun ACM 2011;54(9):69.

Fern N, San I, Cheng KTT. Detecting Hardware Trojans in unspecified functionality through solving satisfiability problems. Proceedings of Asia & South Pacific design automation conference; 2017. p. 598–604.

Fern N, Cheng K. Detecting hardware Trojans in unspecified functionality using mutation testing. Proceedings of IEEE/ACM international conference on computer-aided design; 2016. p. 560–6.

Hu W, Mu D, Oberg J, Mao B, Tiwari M, Sherwood T, et al. Gate-level information flow tracking for security lattices. ACM Trans Des Autom Electron Syst 2014;20(1):1–25.

Jacob N, Heyszl J, Sigl G, Merli D. Hardware Trojans: current challenges and approaches. IET Comput Digit Tech 2014;8(6):264–73.

Jin Y, Makris Y. Hardware Trojans in wireless cryptographic ICs. IEEE Des Test Comput 2010;27(1):26–35.

King J. Symbolic execution and program testing. Commun ACM 1976;19(7):385–94.

Li H, Liu Q, Zhang J. A survey of hardware Trojan threat and defense. Integr: VLSI J 2016;55:426–37.

Microsoft. Z3 – guide; 2017. Available from: http://rise4fun.com/Z3/tutorial/guide

Mirzaei M, Tabandeh M, Alizadeh B, Navabi Z. A new approach for automatic test pattern generation in register transfer level circuits. IEEE Des Test 2013;30(4):49–59.

Ngo XT, Exurville I, Bhasin S, Danger JL, Guilley S. Hardware Trojan detection by delay and electromagnetic measurements. Proceedings of design, automation & test in Europe conference & exhibition, 2015.

Nissim N, Yahalom R, Elovici Y. USB-based attacks. Comput Secur 2017;70:675–88.

Parr T. The definitive ANTLR4 reference. Pragmatic Bookshelf; 2013. p. 328.

Saha S, Chakraborty R, Nuthakki SS, Anshul, Mukhopadhyay D. Improved test pattern generation for hardware Trojan detection using genetic algorithm and boolean satisfiability. Berlin, Heidelberg: Springer; 2015.

Salmani H, Tehranipoor M, Karri R. On design vulnerability analysis and trust benchmarks development. Proceedings of IEEE international conference on computer design (ICCD), 2013.

Shakya B, He T, Salmani H, Forte D, Bhunia S, Tehranipoor M. Benchmarking of hardware trojans and maliciously affected circuits. J Hardw Syst Secur 2017;1(1):85–102.

Shende R, Ambawade D. A side channel based power analysis technique for hardware trojan detection using statistical learning approach. Proceedings of thirteenth international conference on wireless & optical communications networks, 2016.

Takamaeda-Yamazaki S. Pyverilog a python-based hardware design processing toolkit for Verilog HDL. Proceedings of international symposium on applied reconfigurable computing; 2015. p. 451–60.

Tehranipoor M, Koushanfar F. a survey of hardware trojan taxonomy and detection. IEEE Des Test Comput 2010;27(1):10–25.

Waksman A, Sethumadhavan S. Silencing hardware backdoors. Secur Privacy 2011;9(1):49–63.

Wang SJ, Wei J, Huang SH, Li SM. Test generation for combinational hardware Trojans. In: 2016 IEEE Asian Hardware-Oriented Security and Trust(AsianHOST); 2016. p. 1–6.

Xue M, Hu A, Li G. Detecting hardware Trojan through heuristic partition and activity driven test pattern generation. Proceedings of communications security conference; 2014. p. 1–6.

Yi Y, Song H, Yu H, Zhengping R. Study of metamorphic testing. J Converg Inf Technol 2013;8(8):819–27.

Zhang J, Xu Q. On hardware Trojan design and implementation at register-transfer level. Proceedings of IEEE international symposium on hardware-oriented security & trust; 2013. p. 107–12.

Lixiang Shen is currently pursuing the Ph.D. degree from the School of Automation, Northwestern Polytechnical University. She is currently a lecturer with School of Computer Information and Engineering, Changzhou Institute of Technology. And her current research interests include the hardware security, cyber security and risk assessment.

Dejun Mu received the Ph.D. degree in control theory and control engineering from Northwestern Polytechnical University, Xian, Shaanxi, China, in 1994.He is currently a Professor with the School of Automation, Northwestern Polytechnical University, China. His current research interests include control theories and information security, including network information security, application specific chips for information security, and network control systems.

Guo Cao is currently pursuing the Ph.D. degree from the School of Management, Northwestern Polytechnical University. He is currently an associate professor with school of Economics and Management, Changzhou Institute of Technology. And his current research interests focus on the logistics management, theory of decision making.

Maoyuan Qin is a Ph.D. student from the School of Automation, Northwestern Polytechnical University. He received a master degree of automation control from Newcastle University in UK in 2010. He is studying for a doctorate in computer science and technology at Northwestern Polytechnical University. His current research interests include hardware security analysis and formal verification.

Jeremy Blackstone is a Ph.D. student from the Computer Science and Engineering, University of California, San Diego. He received his Bachelor's and Master's degree in computer science from Howard University in Washington, DC. His current research interests include hardware security and fault attacks.

Ryan Kastner is currently a professor in the Department of Computer Science and Engineering at the University of California, San Diego. He received a Ph.D. in Computer Science at UCLA, a masters degree (M.S.) in engineering and bachelor degrees (B.S.) in both Electrical Engineering and Computer Engineering, all from Northwestern University. He leads the Kastner Research Group whose current research interests fall into three areas: hardware acceleration, hardware security, and remote sensing. He is the co-director of the Wireless Embedded Systems Master of Advanced Studies Program. He also co-directs the Engineers for Exploration Program.