Hardware Trojan Detection via Rewriting Logic

Irina Măriuca Asăvoae Ramtine Tofighi-Shirazi
Trusted Labs, Thales Group,
Meudon, France

Adrián Riesco
Universidad Complutense de Madrid,
Madrid, Spain

Uemura Yasuyoshi
Electronic Commerce Security Technology Association,
Tokyo, Japan

Hardware security studies, discovers, and classifies hardware attacks as well as defence strategies such as prevention and protection methods along the entire hardware production chain. Hardware Trojans represent a hardware attack model that emerged in the last decades in the hardware security community. In this paper, we present a methodology for achieving a scalable approach to detect hardware Trojans at the design stage using program transformation in a rewrite-based environment. We evaluate the effectiveness of our methodology on an industrial hardware design, Advanced Encryption Standard cores, which is widely used and deployed for numerous devices and applications.
Keywords: Hardware Trojans, information leakage, Verilog, rewriting logic, programming language semantics specifications, program transformation, symbolic execution.

1 Introduction

The focus in computer systems security was traditionally set on software while the underlying hardware has been considered as trustworthy. However, we witness lately to changes in the practices of the hardware industry such as increased reliance on outsourcing the hardware design. Consequently, the former security trust anchor in hardware became obsolete, so the computer systems security had to split its efforts and focus on hardware security as well.

Hardware security studies hardware attacks as well as defense strategies such as prevention and protection methods along the entire hardware production chain. Hardware Trojans (HT) represent a hardware attack model that encompasses malicious modifications hidden in the hardware with the intention to compromise the designated functionality. For example, an outsourced component may introduce a HT that induces undesired behaviors. HTs may be physically inserted into the hardware during production but could be also introduced in the design as showed in [6] where the cryptographic key is leaked over an antenna or network connection, provided that the correct "easter egg" trigger is applied.

Hardware designs are situated at the beginning of the hardware production chain and were initially handcrafted blueprints of the circuits. The development of the hardware industry introduced the need of automatized ways of production, which determined the apparition of hardware description languages such as Verilog. A Verilog program can be physically realized by synthesis software, which is similar to a Verilog compiler. Namely, synthesis software algorithmically transforms the Verilog program into a netlist–a logically equivalent description consisting only of certain elementary logic primitives–that leads to a circuit fabrication blueprint.

In the last decade HTs became a proactive research domain within the hardware community where the HT problem is studied along the entire hardware production chain [12] and HT benchmarks are proposed [11]. However, up to the date there is no clear evidence of HTs discovered in the wild even though several rumors appeared from time to time, e.g., [10]. Nevertheless, the industry starts to be aware

of this security issue and tries to take measures against it. For example, the work presented in this paper is part of an industrial project commissioned by the New Energy and Industrial Technology Development Organization (NEDO) in Japan with the mission to coordinate and integrate the technological capabilities and research abilities of industry and academia.

Our work proposes a methodology for detection of HTs that produce information leak in the hardware design, i.e., Verilog programs. For this purpose we proceed as follows:

- We give the Verilog program as input to VerilogRLS [8]—a rewriting-based tool built in Maude [4] environment—that specifies the Verilog semantics according to Verilog IEEE standard [2];
- We customize VerilogRLS by defining a program transformation that produces another Verilog program which preserves the data dependencies in the input program;
- We execute symbolically the transformed Verilog program into the customized VerilogRLS to obtain the transitive closure of the data dependency relation for the input program;
- We detect the information leak HTs by using model checking on the data dependency relation.

Our contribution consists in the sound integration of the information leak HT detection for Verilog hardware designs into the rewriting-based specification of the Verilog semantics. This integration comprises a customization of VerilogRLS, which encompasses program transformation and symbolic execution. We also prove here the soundness of the methodology w.r.t. the information leak HT detection, i.e., if the hardware design leaks information during some executions then this is represented into the symbolic execution of the transformed program. Note that, due to the industrial context of the work, the source code is of proprietary nature, which precludes us from making it available.

The structure of this work is as follows: we begin with a brief introduction of Verilog and VerilogRLS in Section 2. Next, in Section 3 we present the program transformation customization of VerilogRLS in two steps: first we define the data dependency relation under consideration and then we describe the program transformation customization of VerilogRLS. In Section 4 we present the symbolic execution we introduce in VerilogRLS, followed by the proof that the symbolic execution of the program transformation is in a simulation relation with the executions of the initial program. In Section 5 we show some experiments of the methodology that we conduct on an industrial size hardware design. A brief related work and conclusions are provided in Section 6.

2 Preliminaries

Verilog is a hardware description language used in the design and verification of digital circuits. A Verilog program can be described by modules executed concurrently and a module consists of a number of program blocks, all executed concurrently. Certain program blocks, e.g., *always* or *continuous assignments*, run continuously while other program blocks, e.g., *initial*, run only once. In Verilog the concurrency is dealt with via a scheduler that separates the concurrent events in five priority ordered categories within which the events are nondeterministically scheduled. VerilogRLS [8] proposes a formal and executable semantics for Verilog language and it is built on top of the rewriting environment provided by Maude[1]. We introduce next the main Verilog elements eventually accompanied by some of their semantics representation specified in VerilogRLS[2].

[1] We assume familiarity with the Maude syntax for rewriting logic. For more details on Maude syntax we refer to [4].

[2] For more details on VerilogRLS we refer to `http://fsl.cs.illinois.edu/index.php/Verilog_Semantics`.

Concurrent processes: Hardware concurrency is modeled in Verilog via three types of syntactic elements: continuous assignment, initial block, and procedural block. A continuous assignment is syntactically designated by `assign v = exp`, where `v` is a variable and `exp` is an expression. The continuous assignment executes whenever some variable in `exp` changes. An initial block, designated by `initial begin Body end`, is executed only once in the beginning of the module containing the block. A procedural block is syntactically designated by `always @(Slist) begin Body end`. The procedural block `Body` is triggered by any update to some variable in the `Slist`, i.e., the sensitivity list. However, we note that the statement in `Body` are executed sequentially while the order of triggers in the sensitivity list does not matter. When multiple events are triggered at the same time, the order of their execution is not specified by IEEE standards except for the order given by the scheduler described next.

Scheduling semantics: A scheduler is an infinite loop that, at each time unit, organizes the concurrent events available for execution. The Verilog scheduler separates the concurrent events in five categories: active, inactive, nonblocking assign updates, monitors, and future events. These categories correspond to priorities (listed here in descending order) the scheduler assigns to each event available at a particular time unit. The events from the same category are scheduled for execution in arbitrary order, which is the main source of nondeterminism in Verilog. The completion of all events in a category at a particular time is called *simulation cycle*. We note that VerilogRLS introduces a subcategory of active events, namely the *listening events*, which are those events that are waiting for some updates to occur in order to get executed. We recall that such an event is produced by the evaluation of an `always` process where the trigger set is designated by the sensitivity list. Hence, a listening event is *activated*, i.e., added to the active evens, as soon there is an update to any of its triggers. VerilogRLS specifies the scheduler in the `SCHEDULING-SEMANTICS` module where each scheduling step is denoted by a rewrite rule.

Assignments: Besides the continuous assignment Verilog provides two other basic types of assignments, blocking and non-blocking, which can only appear in procedural blocks. At their most basic level, the assignments generate update events. The update events themselves are responsible for actually updating the environment of the system and waking up any listening processes. Blocking assignments are syntactically designated as `V = expression ;` with a *blocking* semantics, i.e., once scheduled for execution, any concurrent process is halted until the update event generated by the current blocking assignment completes. VerilogRLS specifies its behavior with equations where the assignment is consumed from the active scheduling category and the change of `V` together with its new value is added to the update category. Note that the update category is further used by the scheduler in a rule for triggering listening events. Non-blocking assignments are designated by the syntax `V <= expression ;`. The non-blocking assignment semantics stipulates that the `expression` is evaluated upon the activation of the assignment but the update of the variable `V` is scheduled in the non-blocking assign update events category. This is specified in VerilogRLS by means of a rule which allows postponing the value change of `V`, and its inclusion in the update scheduling category.

Environment: In VerilogRLS the memory is defined in the module `ENVIRONMENT` as a mapping from data names, of sort `Name`, to values, of sort `Value`. Below we list the VerilogRLS specification of the environment update operator `_[_<-_]`:

```
op _[_<-_] : Env Name Value -> Env .
eq Env[Q <- BV] = insert(Q, [BVToInt(BV) # sizeof(Env[Q])], Env) .
```

where BV is a value of sort BitVector. The format of the BitVector is defined by the constructor operator [_#_] that takes as first argument an integer of sort Int representing the integer value of a bit vector and as second argument a term of sort Nat+, i.e., a natural numbers enriched with infinity, representing the size of the bit vector.

Configuration: A configuration is the term representation for the state of a Verilog program. All equations and rules in the VerilogRLS specification rewrite the configuration to advance the state of the system. The initial configuration of a Verilog program is:

```
env(emptyEnv) time(0) activeProcesses(emptyProcesses) updateEvents(emptyEvents)
nonBlockingAssignUpdateEvents(nilEvents) listeningEvents(emptyEvents)
inactiveEvents(emptyEvents) monitorEvents(emptyEvents)
futureMonitorEvents(emptyEvents) futureEvents(nilEvents)
disables(emptyTrS) output(nilI/O) finish(false)
```

The subterm env represents the environment while time contains the current simulation time used by the scheduler. The configuration subterms activeProcesses, listeningEvents, inactiveEvents, nonBlockingAssignUpdateEvents, monitorEvents, futureEvents contain the event categories used by the scheduler while finish supports the Verilog function $finish that ends an execution.

3 Program Transformation

The goal of the Verilog program transformation presented in this work is to obtain an abstract model which preserves the data dependency of the initial program. We use this abstract model to evaluate the program's vulnerability degree w.r.t. information leakage HTs. In this section we present the program transformation method and its the data dependency fundaments.

```
1  module flipflop_32(dout, din, clk, en, rst);
2   parameter zero = 32'h00000000;
3   input [31:0] din, en;
4   input clk, rst;
5   output [31:0] dout;
6   reg [31:0] dout;
7   always @(posedge clk)
8   begin
9    if (rst) dout <= zero;
10   else if (en) dout <= din;
11  end
12 endmodule
```

```
1  module timebomb(out, in, clock, encrypt, reset);
2   input [31:0] in, encrypt;
3   input clock, reset;
4   output [31:0] out;
5   reg [31:0] tbd;
6   flipflop_32 ff (out, in, clock, encrypt, reset);
7   always @(posedge clock)
8   begin
9    tbd <= tbd + 31'h00000001;
10   if (tbd > 32'hFFFFFFFF) out <= encrypt;
11  end
12 endmodule
```

Figure 1: Illustration of a Verilog program *TBD* implementing a simple time-bomb HT.

Figure 1 shows a Verilog program *TBD* (i.e., To Be Detonated) formed by two Verilog modules that define a flip-flop encapsulated into a time-bomb. The flip-flop module in Figure 1 either resets the output out to zero (provided the rst input is set to true) or transfers the input value din to the output (provided the encryption en is not zero). The time-bomb module in Figure 1 employs the flip-flop in the presence of a timer tbd that is used as a trigger for the leakage of the encryption to the output (in line 10 of time-bomb). The time-bomb is a standardly difficult HT to detect via traditional testing methods due to the fact that the timer is set to a very large value such that the tests fail to reach the triggering point. We use *TBD* as a running example in order to provide the intuition for the program transformation, which we employ for HT information leak detection.

3.1 Data dependency base

The transformation we employ aims to flatten the structure of the program and to maintain only the direct and indirect data dependencies. We define next the data dependency types and we present the program transformation procedure that we employ for the flattening of the program structure.

Let p be a Verilog program. We denote by $A(p)$ the set of the assignment statements appearing in p. Hence, $A(p)$ contains blocking and nonblocking assignments as well as the `assign` statements. Let $a \in A(p)$ be an assignment in p. We denote as $O(a)$ the lefthand side elements of a, i.e., the data *written* by the assignment, and as $I(a)$ the righthand side elements of a, i.e., the data *read* by the assignment. Note that by *data* we understand any segment of a mapping element in the environment operator `env`. Namely, given `x` a mapping element in `env`, a *segment* of `x` is defined as $\texttt{x}[i:j]$ with $0 \leq i \leq j <$`size(x)` two integers. For example, in the non-blocking assignment `out <= encrypt` in line 10 of the `timebomb` module of the *TBD* program in Figure 1, we have $O(a) = \{\texttt{out}\}$ and $I(a) = \{\texttt{encrypt}\}$. Furthermore, we denote by $S(p)$ the sensitivity lists declared in p and by $C(p)$ the conditions in the sequential statements such as if, case, etc. These two sets form the control elements in the Verilog program p in the sense that they determine the execution of other instructions in p. We can define also the I and O sets for the control elements as the data read and respectively written. We note that $\forall t \in C(p) \cup S(p)$ we have $O(t) = \emptyset$ since a control is only reading data for evaluation, e.g., in a sensitivity list l the evaluation is a comparison between the current and previous value of each element in l. For instance, if we consider the sensitivity list in lines 7 in Figure 1, i.e., `@(posedge clk)`, then we have $I(t) = \{\texttt{clk}\}$ and $O(t) = \emptyset$.

Definition 1: We define the two basic types of data dependency that we use as follows:

- The *direct data dependency* of p is the relation $R_d \subseteq D_s \times D_s$ where $(x,y) \in R_d$ *iff* $\exists a \in A(p)$ such that $x \in O(a)$ and $y \in I(a)$;
- The *indirect data dependency* of p is the relation $R_i \subseteq D_s \times D_s$ where $(x,y) \in R_d$ *iff* $\exists c \in C(p) \cup S(p)$ and $\exists a \in Body(c) \cap A(p)$ such that $x \in O(a)$ and $y \in I(c)$, where $Body(c)$ is the `Body` of the statement containing c, e.g., `if` c `Body`;
- The *basic data dependency* of p is the relations $R = R_d \cup R_i$.

Note that the indirect data dependency is usually denoted in the literature as *control dependency*.

The design of the program transformation that we employ aims to produce a Verilog program, which preserves the relation R. In Figure 2 we give the implementation of this program transformation.

The operator `deconstructSet` described in Figure 2 takes a Verilog program statement `Stmt` and, in line `01`, it transforms it into the abstract syntax tree (AST)-term representation via the Maude operator `upTerm`. The equation in lines `02-20` produces recursively the relation `R` from the AST-term representation `OP [ARG, TermL]`, where `OP` is the root of the AST representing a Verilog statement, `ARG` is the first parameter of the statement while `TermL` is the list of the other statement parameters. For example, the AST format of `if (tbd > 32'hFFFFFFFF) out <= encrypt` in line `10` of the `timebomb` module in Figure 1 has `'if'(_')_` as root `OP`, while `ARG` is the condition `tbd > 32'hFFFFFFFF` and `TermL` is the list containing one element, i.e., the body `out <= encrypt` of the if instruction. Each `if` in the equation `02-20` covers a certain class of Verilog instructions. Namely, the `if` in the lines `03-05` produces the `R` relation for assignments, the `if`s in lines `06-09` handle the sensitive lists, the `if`s in lines `10-14` are designated for the branching and the loop instructions, the `if` in lines `15-16` describes the base case of bitvector segments, the `if` in lines `17-18` bypasses the time delays instructions, while in line `19` we have the default case for the rest of the Verilog instructions. The equation in line `21` is a stub for the Verilog instructions not covered by the previous equation, while in the lines `22-25` we have the base case for

```
01: eq deconstructSet(Stmt) = deconstructSet(upTerm(Stmt)) .
02: eq deconstructSet(OP [ARG, TermL])
03:  =  if      OP == '_=_;  or OP == '_=#__;  or OP == '_=@‘(_‘)_;  or OP == '_=@‘(*‘)_;  or OP == '_=@*_;
04:          or OP == '_<=_; or OP == '_<=#__; or OP == '_<=@‘(_‘)_; or OP == '_<=@‘(*‘)_; or OP == '_<=@*_;
05:     then R(downTerm(ARG,deconstructSetExpFailed), deconstructSet(TermL))
06:     else if OP == '@‘(*‘)_  or OP == '@*_
07:     then deconstructSet((ARG, TermL))
08:     else if OP == '@‘(_‘)_
09:     then R(deconstructSet(TermL), deconstructSet(ARG))
10:     else if OP == 'if‘(_‘)_else_  or OP == 'if‘(_‘)_  or OP == 'case‘(_‘)_endcase  or OP == '_:_
11:          or OP == 'repeat‘(_‘)_   or  OP == 'while‘(_‘)_
12:     then R(deconstructSet(TermL), deconstructSet(ARG))
13:     else if OP == 'default‘:_
14:     then deconstructSet((ARG, TermL))
15:     else if OP == '_‘[_:_‘]
16:     then downTerm( OP [ARG, TermL] , deconstructSetFailed)
17:     else if OP == '#__
18:     then deconstructSet(TermL)
19:     else  deconstructSet((ARG, TermL))
20:     fi fi fi fi fi fi fi .
21: eq deconstructSet((ARG, NeTermL)) = deconstructSet(ARG) yy deconstructSet(NeTermL) .
22: eq deconstructSet(C)
23:  = if getType(C) == 'Name
24:   then downTerm(C, deconstructSetFailed)
25:   else emptyCtrS fi .
```

Figure 2: The Verilog program transformation operator `deconstructSet`.

the data, i.e., for registers and wires in lines 23-24 and for constants in line 25. For example, when we apply `deconstructSet` on `if (tbd > 32'hFFFFFFFF) out <= encrypt` it will produce the result `R(R(out, encrypt), tbd yy emptyCtrS)`.

In order to obtain the basic data dependency relation, we have equations that flatten the structure of the `deconstructSet` result such as the equation: `eq R(R(C, EL1), EL2) = R(C, EL1 yy EL2) .` This equation together with the fact that the operator `emptyCtrS` is declared identity element for the `yy` produce the result `R(out, encrypt yy tbd)` when we apply the `deconstructSet` operator on `if (tbd > 32'hFFFFFFFF) out <= encrypt`. Note that, instead of producing pairs for the direct and indirect data dependencies, for each direct dependency pair $(x,y) \in R_d$ we accumulate the indirect dependencies during the `R` flattening with the help of the `yy` operator. Hence, the relation `R` obtained by flattening the result of the `deconstructSet` is defined as:

$$\texttt{R} := \{(x, Y^d \texttt{ yy } Y^i) \mid Y^d = (y^d_j)_{j=1,n}, Y^i = (y^i_j)_{j=1,m} \,:\, \forall 1 \leq j \leq n, (x,y^d_j) \in R_d, \forall 1 \leq j \leq m, (x,y^i_j) \in R_i\}$$

where m,n are two positive integers. Namely, for each assignment $a \in A(p)$ the relation `R` contains a pair $(x, Y^d \texttt{ yy } Y^i)$ with $x \in O(a)$ the output of a, Y^d the list of inputs of a, and Y^i the list of control variables from the indirect dependency pairs R_i related to the assignment a. We observe that `R` is equivalent with the basic data dependency relation R:

$$(x, Y^d \texttt{ yy } Y^i) \in \texttt{R} \textit{ then } (x,y) \in R, \forall y \in Y^d \texttt{ yy } Y^i$$

$$(x,y) \in R \textit{ then } \exists (x, Y^d \texttt{ yy } Y^i) \in \texttt{R} : y \in Y^d \texttt{ yy } Y^i$$

3.2 Program transformation procedure

The relation `R` is the basis for the program transformation we employ for information leakage HT detection. Next we present the program transformation operator.

```
01: eq codeProjection(TL1 assign Stmt TL2) = container(deconstructSet(Stmt)) codeProjection(TL1 TL2) .
02: eq codeProjection(TL1 always Stmt TL2) = container(deconstructSet(Stmt)) codeProjection(TL1 TL2) .
03: eq codeProjection(TL1 Q Q' ( ArgL ) ; TL2) code(IPcode1 module Q (EL) ; TL endmodule IPcode2)
04:  = codeProjection(TL1 replaceArgs(ArgL, EL, TL) TL2) code(IPcode1 module Q (EL) ; TL endmodule IPcode2) .
05: eq codeProjection(TL1 initial Stmt TL2) = initial Stmt codeProjection(TL1 TL2) .
06: eq codeProjection(TL1 parameter Stmt TL2) = parameter Stmt codeProjection(TL1 TL2) .
07: eq codeProjection(TL1 input Decl TL2) = input Decl codeProjection(TL1 TL2) .
08: eq codeProjection(TL1 output Decl TL2) = output Decl codeProjection(TL1 TL2) .
09: eq codeProjection(TL1 inout Decl TL2) = inout Decl codeProjection(TL1 TL2) .
10: eq codeProjection(TL1 reg Decl TL2) = reg Decl codeProjection(TL1 TL2) .
11: eq codeProjection(TL1 wire Decl TL2) = wire Decl codeProjection(TL1 TL2) .
12: eq codeProjection(nilIL) = nilIL .
13: eq container(RSet1) container(RSet2) = container(RSet1 RSet2) .
14: eq container(RSet1) = always @* begin R2Stmt(RSet1) end [owise] .
15: eq R2Stmt(R(X, Y) RSet1) =  X x= Y ;  R2Stmt(RSet1) .
16: eq R2Stmt(emptyRSet) = nilIL .
```

Figure 3: The Verilog program transformation operator `codeProjection`.

Figure 3 contains the equations for the program transformation operator `codeProjection`. The equations `01-02` deposit in the `container` the basic data dependencies, i.e., relation `R`, produced by the `assign` and `always` instructions. The equation `03` replaces a module instantiation with the module body, after the parameters replacement. Finally, the equations `05-11` maintain the `initial` and the declaration instructions as they were as these instructions are used to populate the environment `env` with data. The equation in line `12` is the base case for the `codeProjection` operator while the equations `13-14` regroup all the `R` relations inside the `container`, which is then transformed into an `always @*_` instruction at the end, and the equations `15-16` transform the pairs in the `R` relation into assignments. In order to exemplify the program transformation, we give in Figure 4 the result $\overline{TBD}$ of the `codeProjection` transformation applied to the *TBD* program from Figure 1.

```
1   input [31:0] in, encrypt;
2   input clock, reset;
3   output [31:0] out;
4   reg [31:0] tbd;
5   reg [31:0] out;
6   parameter zero = 0;
7   always @*
8   begin
9    tbd <= {tbd, clock};
10   out <= {encrypt, tbd, clock};
11   out <= {zero, reset, clock};
12   out <= {in, encrypt, reset, clock};
13  end
```

Figure 4: $\overline{TBD}$–the program transformation of *TBD*.

The lines `8` and `9` in $\overline{TBD}$ are generated by the assignments in the lines `9` and respectively `10` in the `timebomb` module in *TBD*. Also, the lines `10` and `11` in $\overline{TBD}$ are generated by the assignments in the lines `9` and respectively `10` in the `flipflop_32` module in *TBD* via its instantiation in line `6` of the `timebomb` module.

Note that the program transformation via `codeProjection` involves a flattening of the Verilog module structure. During the flattening process there can arise name clashes among the variables declared in modules, either for two different modules or for two different instantiations of the same module. To avoid the name clashes, we employ an indexing mechanism for the module instantiation. Namely, upon each module instantiation a unique index is associated to this instantiation and this index is used to rename

the variables in the instantiated module. Then we proceed to the replacement of the formal parameters with the actual parameters. The `codeProjection` operator is employed on the body of the instantiated module, after the variables renaming and parameters instantiation. For brevity, we do not include now the presentation of the module instantiation operator. For this reason, the $\overline{TBD}$ program in Figure 4 does not contain the variable renaming, which is available only for `zero`–the parameter in module `flipflop_32`.

4 Symbolic execution of program transformation

In this section we describe how we adjust VerilogRLS to symbolic execution of Verilog programs and we outline the soundness of the methodology, i.e., the symbolic execution of the program transformation is in a simulation relation with the executions of the initial program.

For a Verilog program p we denote by $\overline{p}$ the program transformation produced by the `codeProjection` operator. We observe that $\overline{p}$ is still a Verilog program with the structure described in Figure 5:

```
IODeclarations : List{ Input | Output | Inout }
DataDeclarations : List{ Reg | Wire }
Initializations : List{ Initial }
always @* begin  RAssigns : List{ BlockingAssign }  end
```

Figure 5: $\overline{p}$ generic structure.

Namely, besides the input/output and data declarations, the transformed program consists of initializations and an `always` block containing the basic data dependencies, i.e., the relation `R`, as blocking assignments.

Since $\overline{p}$ is a Verilog program then $\overline{p}$ can be executed with the VerilogRLS semantics. However, the execution of $\overline{p}$ is potentially unbounded due to the fact that `always @*` is triggered whenever any of the variables in the sensitive list changes value. Note that the sensitivity list generated by `@*` in $\overline{p}$ contains all the righthand side data in the assignment, i.e., $\bigcup_{a \in \texttt{RAssigns}} O(a)$. Also, we recall that the aim of producing $\overline{p}$ is to collect data dependency information for the data in p and we propose to achieve this via symbolic execution of $\overline{p}$ in VerilogRLS. Next we present principles of our implementation of the symbolic execution in VerilogRLS.

In order to transform VerilogRLS into the symbolic version $\overline{\text{VerilogRLS}}$ we first operate changes at the level of the `BitVector` sort in order to incorporate the symbolic values. We construct the sort $\overline{\texttt{BitVector}}$ by adding symbolic values to the first argument of the `BitVector` constructor `[_#_]`. Namely, we replace the `Int` sort of the `[_#_]` by the `SymbolicInt` sort. A symbolic integer, i.e., a term of sort `SymbolicInt`, could be either a standard integer or a `SymbolicValue`, which is constructed as an indexed set of symbolic tokens `SymbolicTokenSet`. Next, the logical and arithmetic operators over `BitVector` need to be abstracted to accommodate the newly introduced symbolic values. For example, the arithmetic operator for equality in $\overline{\texttt{BitVector}}$ is defined as follows:

```
00: var I1 I2 : Int . var ST ST1 ST2 : SymbolicTokenSet . var SIdx SIdx1 SIdx2 : Int .
01:  op _==s_ : SymbolicInt SymbolicInt -> Bool [comm] .
02:  eq I1 ==s I2 = I1 == I2 .
03:  rl (ST $ SIdx) ==s I => false .
04:  rl (ST $ SIdx) ==s I => true .
05:  eq (ST1 $ SIdx1) ==s (ST2 $ SIdx2) = isEqST(ST1, ST2) .
```

We note first that comparing two integers is maintained in `SymbolicInt`, as the equation in line `02` shows. Then, comparing a symbolic value with an integer or another symbolic value can be `true` or

`false`, as seen in the rules in lines `03-04` and, respectively, in the equation in line `05`. The operator `isEqST` compares two sets of symbolic tokens and it behaves as follows: if the two token sets contain only one symbolic token, which is the same in the two set, then the value of `isEqST` is `true`; any other case is handled by rules that render `true` or `false`, nondeterministically, as the rules in lines `03-04` do. Due to this double possible valuation, the $\overline{\text{VerilogRLS}}$ over-approximates the executions of a program $\overline{p}$ as it creates more paths when two data are compared.

For example, after executing the declarations in lines `1-5` in the $\overline{TBD}$ program in Figure 4 the symbolic environment $\overline{\texttt{env}}$ is populated with symbolic values as follows:

```
 in |-> [sy(in) # 32]  encrypt |-> [sy(encrypt) # 32]  clock |-> [sy(clock) # 1]  reset |-> [sy(reset) # 1]
tbd |-> [sy(tbd) # 32]     out |-> [sy(out) # 32]
```

Due to the fact that the `parameter` instruction is only giving a name to a constant and this name never changes, the symbolic environment is going to contain `zero` mapped to the integer `0`. When arriving at the execution of line `6` in Figure 4, the `clock` data, which is in the trigger set of each assignment in the `always` block, is evaluated as possibly different than its previous value. Hence, all the assignments in the lines `8-11` in Figure 4 are executed. After a first iteration of the `always` block the operator $\overline{\texttt{env}}$ contains the following symbolic valuation:

```
  in |-> [sy(in) # 32]  encrypt |-> [sy(encrypt) # 32] clock |-> [sy(clock) # 1]  reset |-> [sy(reset) # 1]
zero |-> [0 # 32]           tbd |-> [sy(tbd) xo sy(tbd) yy sy(clock) # 32]
 out |-> [sy(out) xo sy(encrypt) yy sy(tbd) yy sy(clock) xo sy(reset) yy sy(clock)
                  xo sy(in) yy sy(encrypt) yy sy(reset) yy sy(clock) # 32]
```

We note that `out` contains the `xo` concatenation of all assignments a in the body of `always` with `out` in $O(a)$, i.e., lines `10-12`. However, for the assignment in line `11` `out <= {zero, reset, clock}` only the symbolic values for `reset` and `clock` are maintained int the $\overline{\texttt{env}}$. This is due to the fact that constants such as `zero` are absorbed in our design of symbolic environment. Our aim with this design is to obtain via the symbolic execution of $\overline{p}$ in $\overline{\text{VerilogRLS}}$ the data dependency abstraction for the executions of the entire program p. We note that in our running example $\overline{TBD}$ a second execution of the `always` block renders the same $\overline{\texttt{env}}$ as only the value of `tbd` is replaced into the symbolic valuation of `out` as follows:

```
 out |-> [sy(out) xo sy(encrypt) yy (sy(tbd) xo sy(tbd) yy sy(clock)) yy sy(clock) xo sy(reset) yy sy(clock)
                  xo sy(in) yy sy(encrypt) yy sy(reset) yy sy(clock) # 32]
```

However, the distributivity and the idempotency for the operators `xo` and `yy` render the same symbolic value of `out` as for the previous iteration of `always`. Hence, with this second iteration, we reach the fixpoint from the perspective of the data dependency relation.

4.1 Program transformation invariant preservation

In [5] the authors introduce a language-independent framework for program transformations, which gives a systematic design of syntactic transformations and simpler arguments of their correctness in the abstract interpretation framework. We inherit from [5] their framework but instead of the Galois connection (standard for abstract interpretation) we use the equivalent notion of simulation for proving the soundness of program transformation w.r.t. specific properties, e.g., information leakage.

We show now that for any program p the execution of $\overline{p}$ in $\overline{\text{VerilogRLS}}$ is an abstraction of the executions of p in VerilogRLS, which preserves the data dependency relation. For this purpose we construct a simulation relation between the transition systems $\mathscr{T}$ and $\overline{\mathscr{T}}$ produced by p and $\overline{p}$ in VerilogRLS and $\overline{\text{VerilogRLS}}$, respectively. Note that we rely on the rewriting logic notions of (stuttering) simulations as introduced in [7]. We recall that for two transition systems $\mathscr{T}_1$ and $\mathscr{T}_2$ a stuttering simulation is defined

as the relation S such that for any two states $t_1 \in \mathscr{T}_1$ and $t_2 \in \mathscr{T}_2$ with $(t_1,t_2) \in S$ and any two paths in π_1,π_2 starting in t_1 and t_2, respectively, there exist two strictly increasing functions $\alpha_1,\alpha_2 : \mathbb{N} \to \mathbb{N}$ such that for all $i,j,k \in \mathbb{N}$ if $\alpha_1(i) \leq j < \alpha_1(i+1)$ and $\alpha_2(i) \leq k < \alpha_2(i+1)$ it holds $(\pi_1(j),\pi_2(k)) \in S$. Intuitively, the stuttering simulation allows any of the two paths in the simulation relation to progress between some states, which are given by its associated function α, while the other path may stutter, i.e., remains in the same state.

Definition 2: For any path $\overline{\pi}$ starting in the initial state in $\overline{\mathscr{T}}$ we define the strictly increasing sequence of indexes $(s_i)_{i\in\mathbb{N}}$ such that $\overline{\pi}(s_i)$ is the state where the i-th iteration of the `always` block begins on the path $\overline{\pi}$, for any $i \in \mathbb{N}$. Consequently, the function $\alpha_{\overline{\pi}} : \mathbb{N} \to \mathbb{N}$ with $\alpha_{\overline{\pi}}(i) = s_i, \forall i \in \mathbb{N}$ is strictly increasing.

This definition sets the progress for the executions of $\overline{p}$ between two consecutive iterations of the `always` block. This allows all the data dependencies in p to be once more collected. Then $\overline{p}$ executions continue until no more data dependencies are discovered.

Proposition 1: For any path π starting in the initial state in $\mathscr{T}$ there exists a strictly increasing sequence of indexes $(s_i)_{i\in\mathbb{N}}$ such that any assignment $a \in A(p)$ is executed along the path prefix $\pi(0)..\pi(s_i)$ at most i times. Also, the function $\alpha_\pi : \mathbb{N} \to \mathbb{N}$ with $\alpha_\pi(i) = s_i, \forall i \in \mathbb{N}$ is strictly increasing.

First we remind that any Verilog program produces infinite traces. If there is no assignment in the body of the `always` in $\overline{p}$ then all the assignments in p produce initializations. In this case we set the progress function for p to identity. If there exist assignments that execute infinitely, we set the progress function $\alpha_\pi(i)$ to the state that executes some assignment the i-th time.

Theorem 1:

- There exists a stuttering simulation S between the transition systems $\mathscr{T}$ and $\overline{\mathscr{T}}$ such that the pair of initial states in $\mathscr{T}$ and $\overline{\mathscr{T}}$ is in relation S and for any two paths $\pi,\overline{\pi}$ in $\mathscr{T}$ and $\overline{\mathscr{T}}$, respectively, S is defined based on the functions α_π and $\alpha_{\overline{\pi}}$.
- For any $i \in \mathbb{N}$ we have that the i-th composition of the basic data dependency R^i in the program p is included in the i-th composition of the operator $\mathtt{R}^i$ defined for $\overline{p}$ on the path segments $\pi[\alpha_\pi(i)..\alpha_\pi(i+1)-1]$ and $\overline{\pi}[\alpha_{\overline{\pi}}(i)..\alpha_{\overline{\pi}}(i+1)-1]$, where a path segment $\pi[i_1,i_2], i_1 \leq i_2 \in \mathbb{N}$ is the sequence of states $\pi(i_1)..\pi(i_2)$.

The definition of the relation S follows the prerequisites of the stuttering simulation based on the progress functions from Definition 2 and Proposition 1. Namely, for any two paths $\pi,\overline{\pi}$ in $\mathscr{T}$ and $\overline{\mathscr{T}}$, respectively, and for any $i,j,k \in \mathbb{N}$ such that $\alpha_\pi(i) \leq j < \alpha_\pi(i+1)$ and $\alpha_{\overline{\pi}}(i) \leq k < \alpha_{\overline{\pi}}(i+1)$ we have $(\pi(j),\overline{\pi}(k)) \in S$. In other words, two paths segments given by two consecutive progression points, i.e., values of the α functions for i and $i+1$, are in the stuttering simulation relation S. Furthermore, we prove by induction that the simulation relation S is sound w.r.t. the data dependency relation R. This entitles us to evaluate the information leakage on the executions of the program $\overline{p}$ and to report the results as *potential HT* on p. Note that the construction of the simulation relation may provide bound information in the context of a concrete evaluation of the information leakage HTs.

Proposition 2: There exists $t \in \mathbb{N}$ such that $\mathtt{R}^* = \mathtt{R}^t$, i.e., the transitive closure of `R` is calculated after a finite number t of iterations in $\overline{\texttt{env}}$.

The proof of this proposition relies on the fact that there are finitely many data dependency pairs that can be collected in `R`. Consequently, given that $\mathtt{R}^t$ is collected in $\overline{\texttt{env}}$, the executions in $\overline{\mathscr{T}}$ keep $\overline{\texttt{env}}$ unchanged after a finite number of steps.

Theorem 2: The executions of the program $\overline{p}$ in $\overline{\text{VerilogRLS}}$ contain in $\overline{\texttt{env}}$ an over-approximation of the transitive closure of the data dependency relation in the original program p. Moreover, this approximation can be effectively evaluated.

The proof of this theorem is based on combining the results of Theorem 1 and Proposition 2. Namely, the stuttering simulation provides the over-approximation result via the connection between the data dependencies in p and the relation R while Proposition 2 gives the connection between the relation R and the $\overline{\texttt{env}}$ obtained on the execution paths prefixes of $\overline{p}$. Note that we skip several reasoning steps between Theorem 1 and Theorem 2. These together with complete proofs are to be detailed at a later stage.

Observation 1: The detection of information leak HTs can be obtained from the transitive closure of the data dependency relation via the existence of dependencies between the secret and the output data.

Observation 2: Note that a more precise detection of information leakage HTs, i.e., reduction of false positives introduced by the over-approximation, could be obtained in several ways:

- Enriching the data dependency relation with additional information. Namely, the assignments in $\overline{p}$ could collect constraints representing the arithmetic expressions given by the direct data dependencies in p or the logical expressions given by the indirect data dependencies in p. In this case the updates in $\overline{\texttt{env}}$ require the intervention of an SMT-solver.
- Enriching $\overline{p}$ with tracing information about the locations in p used to construct each assignment in $\overline{p}$ and adding this information to the updates in $\overline{\texttt{env}}$. Hence, we can reconstruct from $\overline{\texttt{env}}$ the slice of the initial program involved in the reported information leakage. To verify the accuracy of the report, the slice of p can be model checked in VerilorRLS, i.e., in the concrete semantics, to either validate or refute the detection of information leakage.
- Enrich the information leakage detection with metrics on the degree of data leakage, which measure the quantity of bitvector segments being leaked. Based on these metrics, a warning may be discarded if the amount of information leakage is considered insignificant.

5 Experiments

In this section we first introduce our industrial use-case, which we used to conduct experiments, then we describe the HT patterns we employ and the experimental results we obtained on the use-case and its variations with HT insertions.

The evaluation of our program transformation tool is made on the Advanced Encryption Standard (AES) hardware design, a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001 [1]. The AES Verilog code contains 37 modules of approximatively 4000 lines of code. The industrial usage of AES hardware design is wide, aiming to achieve data privacy and authenticity in many applications. Next we give a non-exhaustive list of existing applications:

- Electronic financial transactions: eCommerce, banking or point-of-sale.
- Secure corporate communications: for Storage Area Networks (SAN), Virtual Private Networks (VPN), or video conferencing.
- Personal mobile communications: PDA, Wearables, Point-to-Point Wireless.
- Secure environments: satellite communication, network appliances or surveillance systems.

The extensive usage of the AES, its inherent complexity, and the sensitivity of the computations it involves, makes this industrial use-case relevant for the evaluation of our methodology towards the detection of HTs. The evaluation of our methodology is made on the AES Verilog code, with and without HT patterns. The HT patterns experimented with are the followings:

- *Permanent key leakage*: Every time plain text is ciphered, 10 clock cycles after the highest byte of the plain text has been loaded, the key is leaked on the ciphered text output. The leak occurs during 4 consecutive clock cycles, corresponding to the four bytes of the key. After that, the computation becomes normal again.
- *Loadable key flag*: During the encryption of message `#n`, the `KSTAT`[3] output is asserted if and only if the bit of index `n[128]` of the key is 1. Otherwise `KSTAT` is de-asserted.

The results of our evaluations are illustrated in Table 1. We used Maude v2.7 on a laptop running Ubuntu 18.04 with 8 GB of RAM and a Intel Core i7-6820HQ vPro processor.

Module	Data leak detection	Time of execution
`aescore-clean`	False	4m18,341s
`aescore-permanent-key-leak`	True	2m47,054s
`aescore-load-key-flag`	True	1m57,542s

Table 1: Evaluation results on our industrial use-case

Note that the information leak detection makes a search in the data dependency relation collected in $\overline{\texttt{env}}$ to discover if there are connections between the secret data, e.g., the encryption key, and the output data. It so happens that the permanent key leakage and loadable key flag produce a singleton dependence between the key and the output, which we know that it comes from an assignment or a condition in the original program. However, if we introduce a security metric, which says that only one bit leak is not considered a security threat, then the loadable key flag is a false positive. In order to evaluate Trojans models coverage, more experiments may be done on different use-cases and/or other HT models based on the benchmarks provided at `https://www.trust-hub.org/home`.

6 Conclusions and Related Work

In this work we introduced the methodology for achieving a scalable approach to detect information leak hardware Trojans at the design stage using a program transformation and symbolic execution applied over the VerilogRLS tool, the rewriting based semantics specification of the Verilog language. We exemplified the methodology on an industrial-scale hardware design used for encryption, which allows us to evaluate the scalability of our methodology for industry.

Related work: The HT detection using information flow analysis is presented in [9]. The method relies on an the adaptation of an automatic test pattern generator (ATPG) algorithm to propagate the fault in order to identify the points through which the secret can be leaked, directly or indirectly. The drawback of this approach is that ATPG does not provide a definite (hence exact) answer due to time-out restrictions.

Moreover, the synthesis tools, which transform the hardware design into RTL design and then into the gate-level representation, introduce various optimizations upon each transformation. For example,

[3] Key port status. When Asserted, loading of cipher keys is not allowed.

the resource sharing optimization introduce imprecision into Information Flow Tracking (IFT), because the tracking rules define only approximations of the real flow. Consequently, the authors in [3] propose to gain precision in IFT by simply lifting the static analysis resolution at the RTL level where, presumably, the IFT is more performant, *i.e.*, faster and more precise.

Acknowledgements: This paper is based on results obtained from a project, JPNP18015, commissioned by the New Energy and Industrial Technology Development Organization (NEDO). Adrián Riesco's work was supported by Comunidad de Madrid as part of the program S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union.

References

[1] (2001): *Specification for the Advanced Encryption Standard (AES)*. Federal Information Processin Standards Publication 197.

[2] (2006): *IEEE Standard for Verilog Hardware Description Language*. IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001).

[3] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen & Ryan Kastner (2017): *Register transfer level information flow tracking for provably secure hardware design*. In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, IEEE, pp. 1691–1696.

[4] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & Carolyn L. Talcott, editors (2007): *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. *Lecture Notes in Computer Science* 4350, Springer.

[5] Patrick Cousot & Radhia Cousot (2002): *Systematic design of program transformation frameworks by abstract interpretation*. In: *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, ACM, pp. 178–190.

[6] Wei Hu, Baolei Mao, Jason Oberg & Ryan Kastner (2016): *Detecting Hardware Trojans with Gate-Level Information-Flow Tracking*. *IEEE Computer* 49(8), pp. 44–52.

[7] Narciso Martí-Oliet, José Meseguer & Miguel Palomino (2008): *Algebraic Stuttering Simulations*. *Electron. Notes Theor. Comput. Sci.* 206, pp. 91–110.

[8] Patrick O'Neil Meredith, Michael Katelman, José Meseguer & Grigore Rosu (2010): *A formal executable semantics of Verilog*. In: *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), Grenoble, France, 26-28 July 2010*, IEEE Computer Society, pp. 179–188.

[9] Adib Nahiyan, Mehdi Sadi, Rahul Vittal, Gustavo K. Contreras, Domenic Forte & Mark Tehranipoor (2017): *Hardware trojan detection through information flow security verification*. In: *IEEE International Test Conference, ITC 2017, Fort Worth, TX, USA, October 31 - Nov. 2, 2017*, IEEE, pp. 1–10.

[10] Jordan Robertson & Michael Riley (2018): *The Big Hack: How China Used a Tiny Chip to Infiltrate Amazon and Apple*. *Bloomberg Businessweek* 4th of October, p. 1. Available at `https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies`.

[11] Bicky Shakya, Miao Tony He, Hassan Salmani, Domenic Forte, Swarup Bhunia & Mark Tehranipoor (2017): *Benchmarking of Hardware Trojans and Maliciously Affected Circuits*. *J. Hardware and Systems Security* 1(1), pp. 85–102.

[12] Kan Xiao, Domenic Forte, Yier Jin, Ramesh Karri, Swarup Bhunia & Mark Mohammad Tehranipoor (2016): *Hardware Trojans: Lessons Learned after One Decade of Research*. *ACM Trans. Design Autom. Electr. Syst.* 22(1), pp. 6:1–6:23.