Hardware Trojans Hidden in RTL Don't Cares - Automated Insertion and Prevention Methodologies

Nicole Fern, Shrikant Kulkarni and Kwang-Ting (Tim) Cheng
University of California - Santa Barbara
ECE Department
Email: {nicole, skulkarni, timcheng}@ece.ucsb.edu

Abstract—**Don't cares in RTL code have long plagued chip verification due to hard-to-diagnose "X-bugs" resulting from ambiguous X simulation semantics, yet prevail in modern designs because of enormous opportunities for area/performance/power optimization during synthesis. We analyze don't cares specified at the RTL level from a security perspective and propose a novel class of Hardware Trojans which leak internal circuit node values using only existing design don't cares. Detection of this Trojan class is impossible using either functional simulation/verification or a perfect sequential equivalence checker. We then provide a formal automated X-analysis technique which both prevents the insertion of this new Trojan type and also has the potential to uncover accidental X-bugs as well. We provide several examples, including an Elliptic Curve Processor, illustrating both Trojan insertion and our prevention technique.**

I. Introduction

A. Hardware Trojans

Addressing the threat of hardware Trojans is becoming a priority for both semiconductor design houses and the U.S. government [7]. The design, manufacturing, testing, and deployment of silicon chips involves many companies and countries. If a single party involved deems it advantageous to insert malicious functionality into the chip, referred to as *Hardware Trojans*, the consequences can be catastrophic.

Hardware Trojans may be inserted into the RTL code, gate-level netlist, circuit layout, or circuit mask for a given design. Examples of malicious functionality accomplished by Trojan circuitry range from denial of service attacks such as premature aging and bus deadlock to subtler attacks which attempt to gain undetected privileged access on a system, leak secret information through side channels, or weaken random number generator output [23].

Many Trojan taxonomies have been proposed [23], [15], [27], which categorize Trojans based on the design phase they are inserted, the triggering mechanism, and malicious functionality accomplished (payload). Most existing Trojans can be divided into the following categories:

1) The logic functions of some design signals are altered, causing the circuit to violate the system specification
2) The Trojan leaks information through side-channels, and no functionality of any existing signals is modified

Our work addresses a third, less studied type of Trojan:

3) The logic functions of **only** those design signals which have **unspecified behavior** are altered to add malicious functionality without violating system specifications

The key difference between Categories 2 and 3 is that Trojans in Category 3 alter the design in the boolean/functional domain, whereas Trojans in Category 2 only manipulate non-boolean side channels, and require characterization of these side channels for detection.

In this work, the unspecified behavior necessary to implement Trojans in Category 3 results from **don't cares specified by the designer in the RTL code**. We present techniques to both insert and prevent insertion of Category 3 Trojans in the RTL code or gate-level netlist.

An attacker can assign values or tie other internal design signals to RTL don't cares to accomplish malicious functionality, such as leaking secret information. Prevention of this new Trojan type requires **refining the system specification** (Verilog code) by first identifying the "dangerous" don't cares, then disambiguating them by selecting static values to assign.

Many of the Trojans proposed in literature claim to hide in unspecified design functionality, but actually hide from the verification effort using extremely rare triggering conditions. Examples of stealthy Trojan triggers are counters, which wait millions of cycles before changing circuit functionality, or pattern recognizers, which wait for a "magic" value or sequence to appear on the system bus [26] or as plaintext input in cryptographic hardware [9], [11], [8]. Trojans with these triggering mechanisms generally deploy a payload which clearly changes the circuit functionality and thus violates system specifications (such as causing a fault during a cryptographic computation). Therefore, such Trojans are Category 1 Trojans.

Unlike Trojans which rely on rare triggering conditions to avoid causing incorrect design behavior during testing and normal design operation, our proposed Trojans are theoretically impossible to detect even if all possible input sequences are tested.

Moreover, our Trojans are undetectable even if a perfect sequential equivalence checker is used to check if a Trojan infested RTL or gate-level implementation conforms to a golden RTL design. This is because the design behavior being maliciously modified by the Trojan payload is itself unspecified in the original specification! Therefore, existing pre-silicon detection methodologies targeting identification of nearly unused circuitry, or rare node values [22], [29] do not address this new Trojan type.

IP watermarking by embedding secret information in the assignment of don't care values [20] is conceptually similar to the proposed Trojan insertion methodology, as both view RTL don't cares as an opportunity for the insertion of extra functionality.

To the best of our knowledge, [17] is the only work which recognizes the potential to implement malicious behavior in unspecified design functionality. [17] defines unspecified functionality as incompletely specified state transition and output functions, given a digital system specified as a finite-state machine (FSM). The process of logic synthesis takes an incompletely specified FSM M and transforms M to a completely specified gate-level FSM, M', which may contain additional unwanted state transitions and output assignments while still conforming to the original FSM.

The method proposed in [17] uses state reachability as a metric for trust. First the designer must manually categorize all design states as either protected or non-protected in a golden symbolic FSM model (M). If a path to a protected state exists in the gate-level implementation (M'), but does not exist in M, M' is considered untrusted.

Our work differs significantly from this approach and overcomes the following limitations of the state reachability based method for machines specified using symbolic FSM models proposed in [17]: 1) analysis must be performed on a *symbolic* representation of the design state space, 2) the labeling of protected v. non-protected symbolic states must be done manually and it is likely most designers would not have a clear idea or guidance for this labeling task, and 3) either full reachability analysis of protected states is required, making the method unscalable to modern designs, or the T flip-flops in the circuit can be modified so *no* transitions from unprotected to protected states are allowed.

While it is common for protocols and controller modules to have reference state diagrams or state tables, from which a symbolic representation can be built and analyzed, often the only available specification for a complex design before logic synthesis is described in HDL such as the RTL code. We focus only on analyzing RTL don't cares since these precisely represent the freedom given to the synthesis tool for optimization and the freedom available to the attacker for implementing malicious functionality.

The second major difference between our work and [17] is that our notion of dangerous unspecified functionality is based on information leakage potential instead of protected state reachability. This has the main advantage that the designer is only required to identify attacker-observable signals, avoiding the high-effort manual categorization of symbolic states as protected or non-protected.

B. RTL X's and Don't Cares

X's appearing in RTL code have different semantics for simulation and synthesis. In RTL simulation, an X represents an unknown signal value, whereas in synthesis, an X represents a don't care, meaning the synthesis tool is free to assign the signal either 0 or 1.

During RTL simulation there are two possible sources of X's: 1) X's specified in the RTL code (either explicitly written by the designer or implicit such as a case statement with no default), and 2) X's resulting from uninitialized or undriven signals, such as flip-flops lacking a known reset value or signals in a clock-gated block. X's from source 1 are don't cares, and are assigned values during synthesis, meaning they are *known* after synthesis, whereas X's from source 2 may be unknown until the operation of the actual silicon.

The Trojans we propose take advantage of source 1 X's, and clearly, if the design logic is fully specified, and don't cares never appear in the Verilog code, these Trojans cannot be inserted. However, don't cares have been used for several decades to minimize logic during synthesis [10], and forbidding their use can lead to unacceptable area/performance/power overhead. For the case study presented in Section IV, replacing all X's in the control unit Verilog with 0's results in almost an 8% area increase for the block.

[24] and [19] give an industry perspective and overview of the many problems caused by RTL X's during chip design/verification/debug along with a survey of existing techniques and tools which address X-issues. Simulation discrepancies between RTL and gate-level versions of a design due to X-optimism and X-pessimism, and propagation of unknown values due to improper reset or power management sequences [16] are all issues addressed by existing research and commercial tools.

Our work presents yet another issue resulting from the presence of X's in RTL code, and provides further incentive to allocate verification resources to these existing X-analysis tools. However, existing tools aim to uncover *accidental* functional X-bugs, while the Trojans we propose can be considered a special pathological class of X-bug specifically crafted with malicious intent to avoid detection during functional verification.

This means that X-analysis tools which focus only on providing RTL simulation with accurate X semantics, perform X-propagation analysis only for scenarios occurring during simulation-based verification, or formal methods which only analyze a limited number of cycles (ex. the reset sequence) do not adequately address the proposed threat. Through the examples in the remainder of the paper we aim to highlight the aspects of this new threat that differ most from the existing X-bugs targeted by commercial and academic tools.

The rest of the paper is organized as follows: Section II states the threat model we are addressing, and presents 2 simple examples to illustrate how typical usage of don't cares in Verilog code can potentially lead to undesired information leakage, Section III presents an automated methodology to analyze all don't care bits in a Verilog design and classify them based on their information leakage potential, Section IV shows the application of this methodology to an Elliptic Curve Processor design with several hundred don't care bits, and Section V summarizes our contributions.

II. Defining Malicious Don't Cares

A. Threat Model

The Trojans we are proposing are inserted at RTL or gate-level with the aim of avoiding detection by equivalence checking against a Trojan-free RTL model.

Equivalence checking at the RT level of abstraction becomes *conformance* checking in the presence of RTL don't cares, because a single RTL specification simultaneously represents several possible valid gate-level implementations.

When performing equivalence checking between an RTL and gate-level implementation, the gate-level implementation needs to match *only one* of all possible valid gate-level implementations specified by the RTL [18]. The proposed Trojans take advantage of this by transforming the design into a malicious, but valid implementation.

The Trojans can be inserted by a malicious CAD tool, disgruntled employee, or any person with access to the RTL code and the ability to modify either the RTL or the netlist.

Our prevention methodology assumes the existence of a Trojan-free RTL model to perform X-analysis on, and provides an improved Trojan-free model that can be used detect any Category 3 Trojan through equivalence checking or simulation-based verification methods. Although requiring the existence of a Trojan-free RTL model may seem limiting, one should remember that before an attacker can successfully insert a Trojan by defining RTL don't cares there must exist a Trojan-free version of the RTL code containing the X's that the attacker hopes to exploit.

B. Illustrative Examples

Example 1: To illustrate how don't cares can be exploited to perform malicious functionality, a contrived example is presented for illustrative purposes. The module given in Listing 1 transforms a 4-bit input by either inverting, XORing with a secret key value, or passing the data to the output unmodified. The choice between the 3 transformations is selected using a 2-bit control signal, `control`. When `control=11`, Line 17 specifies that `tmp` can be assigned any value by the synthesis tool to minimize the logic used.

Listing 1: simple.v

```
1  module simple (clk,reset,control,data,key,out);
2  input clk, reset;
3  input [1:0] control;
4  input [3:0] data, key;
5  output reg [3:0] out;
6  reg [3:0] tmp;
7  //tmp only assigned a meaningful value
8  //if control signal is 00, 01 or 10
9  always @ (*) begin
10   case(control)
11     2'b00: tmp <= data;
12     2'b01: tmp <= data ^ key;
13     2'b10: tmp <= ~data;
14   //Trojan logic ————
15   //2'b11: tmp <= key;
16   //——————————
17     default: tmp <= 4'bxxxx;
18   endcase
19 end
20 always @ (posedge clk) begin
21   if (~reset) out <= 4'b0;
22   else out <= tmp;
23 end
24 endmodule
```

An attacker can take advantage of the implementation freedom given by the RTL by assigning `key` to `tmp`, causing the secret key value to appear at the output of this module. The Trojan can be inserted in the RTL code by uncommenting Line 15, or at gate-level by modifying the netlist after synthesis.

It should be emphasized that in either case, since the assignment of `tmp` during the `control=11` condition is **unspecified**, it is impossible to detect the Trojan even if the design can be exhaustively simulated, or a perfect equivalence checker can compare the golden and Trojan implementations. As an example, Cadence Conformal LEC [3] was used to perform 2 experiments: equivalence checking between Golden RTL and Trojan RTL, and equivalence checking between Golden RTL and a Trojan infested netlist. In both cases, the equivalence checker was unable to detect the presence of the Trojan functionality.

It should also be noted that the don't cares assigned to `tmp` in Line 17 are *useful* to the attacker because:

1) The don't care assignment is reachable
2) A primary output (which the attacker can observe) differs depending on the value of the don't care bits

Example 2: In the previous example, all the don't care bits are dangerous and should be disambiguated in the RTL code. The following example (similar to Example 1 with the addition of a 3-bit FSM with 5 reachable states) illustrates that not all don't cares are dangerous, and that the goal of any Trojan prevention or X-analysis technique is to identify only the dangerous X's and allow the synthesis tool to use the remaining don't cares for logic minimization.

Listing 2: simple_state.v

```
1  module simple_state(clk,reset,control,data,key,out);
2  input clk, reset;
3  input [1:0] control;
4  input [3:0] data, key;
5  output reg [3:0] out;
6  reg [3:0] tmp;
7  reg [2:0] counter, next_counter;
8  reg [3:0] pattern;
9  //Truncated Counter 0–4
10 //5,6, and 7 never appear
11 always @(*) begin
12   if (counter < 3'h4)
13      next_counter <= counter + 3'b1;
14   else next_counter <= 3'b0;
15 end
16 always @(posedge clk) begin
17   if (~reset) counter <= 3'b0;
18   else counter <= next_counter;
19 end
20 always @(*) begin
21   case(counter)
22     3'd0: pattern <= 4'b1010;
23     3'd1: pattern <= 4'b0101;
24     3'd2: pattern <= 4'b0011;
25     3'd3: pattern <= 4'b1100;
26     3'd4: pattern <= 4'b1xx1;
27     default: pattern <= 4'bxxxx;
28   endcase
29 end
30 always @ (*) begin
31   case(control)
32     2'b00: tmp <= data;
33     2'b01: tmp <= data ^ key;
34     2'b10: tmp <= ~data;
35     2'b11: tmp <= data ^ {pattern[3], pattern[2:0] &
         counter};
36   endcase
37 end
38 always @ (posedge clk) begin
39   if (~reset) out <= 4'b0;
40   else out <= tmp;
41 end
42 endmodule
```

In Listing 2 there are 6 total assignments of 1-bit don't care values. One could replace these X's in the Verilog code with 6 1-bit signals, $dc_0, dc_1, ..., dc_5$. The attacker can then choose to assign other internal design signals (such as key bits) to the don't care bits or leave them for the synthesis tool to assign. Line 27 can be re-written as:

$$\texttt{default: pattern <= }\{dc_0\texttt{, } dc_1\texttt{, } dc_2\texttt{, } dc_3\}\texttt{;}$$

Line 27 is unreachable (and thus `pattern` will never be assigned $dc_0 - dc_3$) because the variable `counter` only takes on values 0-4. These X's are safe, and cannot be used to leak information, therefore are best left in the RTL to aid in logic optimization. A more interesting X-assignment occurs on Line 26, which can be re-written as:

$$\texttt{3'd4: pattern <= }\{\texttt{1, } dc_4\texttt{, } dc_5\texttt{, 1}\}\texttt{;}$$

The assignment of $\{dc_4\texttt{, } dc_5\}$ to `pattern[2:1]` is reachable, however, by manual inspection, one can see that the only assignment influenced by `pattern` (Line 35) contains a bitwise AND between `counter` and `pattern[2:0]`, which prevents dc_5 from propagating further, but not dc_4! This is because when `counter = 3'd4`, Line 35 effectively becomes:

$$\texttt{2'b11: tmp <= data \^{} }\{\texttt{1, } dc_4\texttt{, 0, 0}\}\texttt{;}$$

In this example only 1 of 6 don't cares is dangerous and necessary to remove. In a design with hundreds of don't cares, it is expected that only a small subset is dangerous, which motivates why it is valuable for an X-analysis tool to take a fine-grain approach and distinguish between unreachable, reachable but non-propagating don't cares, and don't cares that have the potential to propagate to outputs or attacker observable points.

C. Formal Definition

The following sets of signals can be defined for any design:

S: all signals
D: don't care bits in the RTL code, where $D \subseteq S$
I: signals an attacker can influence, where $I \subseteq S$, and $D \subseteq I$
O: signals an attacker can observe, where $O \subseteq S$

The following sets are defined for each dc_i, $dc_i \in D$:

O_i: observable signals which differ when $dc_i = 0/1$, $O_i \subseteq O$
P_i: set of primary input sequences (starting from design reset) which cause signals in O_i to differ

Sets O and I can be determined based on the design specification, but it is reasonable to assume that O consists of all primary outputs and I consists of all primary inputs and don't care bits.

Through the examples in the previous section, we have seen that a don't care bit, dc_i, is dangerous **iff** $P_i \neq \emptyset$. We will call an input sequence, $T_i \in P_i$, a **distinguishing input sequence** for dc_i. Our solution for identifying dangerous don't cares, given in Section III, determines if $P_i \neq \emptyset$. For scalability reasons, our solution may over-approximate the set of don't cares classified as dangerous.

If dc_i is classified as dangerous, dc_i should be specified in the Verilog code, instead of being left as a don't care bit. If the circuit designer cannot afford to specify all dangerous don't cares due to tight area and timing constraints, the following metric based on $|P_i|$ provides a ranking that can be used to replace the don't cares most accessible to an attacker.

$|P_i|$ reflects the information leakage potential of dc_i because if $|P_i|$ is large, then there exist many conditions under which the attacker can learn the value of dc_i. Considering the probability of each sequence in P_i occurring during normal circuit operation, and how many sequences the attacker can force to occur can also improve the accuracy of this metric. An attacker can force a distinguishing input sequence T_i, if all signals in T_i are in I.

III. Automated Identification of Dangerous Don't Cares

A. Methodology

The problem of finding if a distinguishing input sequence exists has been formulated in [25] as a sequential equivalence checking problem. In [25], the analysis was performed to find X-bugs, not prevent Hardware Trojans, but like the Hardware Trojans we are proposing, X-bugs result from reachable X-assignments that affect primary outputs in the design.

One key difference between X-bugs and the proposed Trojan type is that in many designs, for example, a serial multiplier, or the Elliptic Curve Processor analyzed in Section IV, the values at the primary outputs of the unit during intermediate cycles in the computation typically don't matter, as long as the final computation result is correct. X's propagating to primary outputs during intermediate cycles generally aren't considered X-bugs if the final result is unaffected, however, information leakage can still occur during these intermediate cycles if the attacker can observe the primary outputs of the circuit.

The equivalence check is performed between 2 near identical versions of the design: one where $dc_i = 0$ and one where $dc_i = 1$. If the designs are identical under all possible input sequences ($P_i = \emptyset$), dc_i cannot possibly be used to leak design information.

We build upon this idea further by addressing the relationship between multiple don't cares in the design, and we formulate the problem in terms of combinational equivalence checking and state reachability analysis.

While combinational equivalence checking between two nearly identical designs is efficient and scalable, state reachability analysis is not. In the Elliptic Curve Processor case study presented in Section IV, we illustrate how commercial code-reachability tools can be used in place of symbolic state reachability analysis to re-classify don't cares erroneously marked as dangerous after combinational equivalence checking as safe.

Consider the generic example circuit in Figure 1, where the sequential behavior has been removed by making all flip-flop inputs pseudo primary outputs (PPOs) and all flip-flop outputs pseudo primary inputs (PPIs). There are n don't care bits in the design, and it is clear that dc_i and dc_j have the ability to block each other from propagating. dc_h is in the fan-in cone for signal a, and can also influence the propagation of dc_i and dc_j, while dc_k is completely independent from dc_i, dc_j, and dc_h.

Combinational equivalence checking can be performed between 2 versions of the original design: $C_{dc_i=0}$, and $C_{dc_i=1}$, by constructing the miter in Figure 2 and checking the satisfiability of node z. If z is UNSAT, then dc_i is safe. Otherwise, the

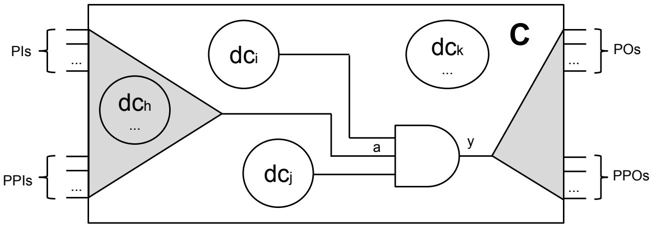

Fig. 1: Generic Circuit with Don't Care Bits

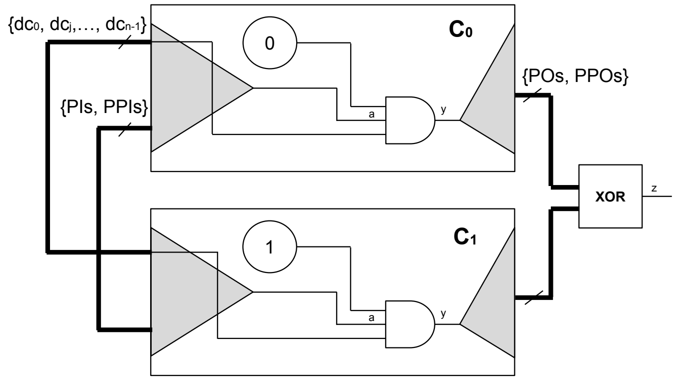

Fig. 2: Equivalence Checking Formulation

equivalence checker returns a distinguishing input vector (since we are performing combinational analysis, the distinguishing sequence is now just a single vector).

Note that when analyzing dc_i, all remaining $n-1$ don't care bits are made primary inputs. This ensures the distinguishing input vector contains information about how the remaining don't care bits are constrained if dc_i is to successfully leak information.

Since we are not considering the sequential behavior of the design, the distinguishing input vector could require that the pseudo primary inputs be assigned a value that can never occur, in other words an *unreachable state*. State reachability analysis can be performed before analysis of all don't care bits, and a logic formula describing the set of unreachable states can be incorporated into the miter circuit to prevent the equivalence checker from finding distinguishing input vectors containing these states.

State reachability is a hard problem, but recent advances in model checking [13] and techniques such as [14], which over-approximate the set of reachable states, can aid in addressing non-trivial designs. Additionally, since don't cares can often be traced back to single-line assignments in the Verilog code, dead-code analysis and code reachability tools can help easily eliminate don't care assignments that are unreachable.

For Trojan prevention, an over-approximation is ideal because it ensures that a dangerous don't care will never be classified as safe due to the elimination of a distinguishing input vector containing a state erroneously marked as unreachable.

Our analysis uses the robust Verilog parsing capabilities of Yosys [28] to identify don't care bits in the design, and create and write $C_{dc_i=0}$, and $C_{dc_i=1}$ in the Berkeley Logic Interchange Format (BLIF). The logic synthesis tool ABC [1] is then used to perform combinational equivalence checking using the `cec` command. ABC is also used to compute the set of unreachable design states (which is possible for the toy examples in Section II-B) using the `ext_seq_dcs` command.

B. Existing X-Analysis Tools

Our experiments use ABC and Yosys because of their public availability and transparency, however we are aware that many commercial X-analysis tools and formal engines exist with the capability to perform similar analysis, such as Jasper X-prop [5], Atrenta Spyglass [2], Cadence Incisive [4], and Synopsys Magellan [6], to name only a few.

Our intention is not to argue that the existing tools are incapable of performing the necessary analysis but that settings do not exist in these tools for analyzing don't cares in a security context. We illustrate our approach in the general terms of equivalence checking and state reachability to provide a clear guide to be used by others looking to extract the same information by taking advantage of access to existing commercial tools with advance debug capabilities and optimized runtimes.

C. Methodology Applied to Examples 1 and 2

Our tool correctly classifies all 4 don't care bits in Example 1 as dangerous. If Example 2 is analyzed without state reachability analysis, our tool classifies all don't cares as dangerous except for dc_5, which is classified as safe due to the bitwise AND on Line 35 which always prevents propagation. All the counterexamples for $dc_0 - dc_3$, assign `counter` to 101, which can never occur.

ABC is used to perform reachability analysis and describe the forbidden states as a logic function, L, which takes as input all pseudo-primary inputs and outputs 1 if the input combination can never occur. We then modify the miter circuit in Figure 2 by adding an extra XOR gate, with inputs L' and z. If the modified network is satisfiable, then the don't care is dangerous.

Augmenting our methodology to include state reachability information results in the correct categorization of $dc_0 - dc_3$ as safe, leaving only dc_4 classified as dangerous.

IV. Elliptic Curve Processor

We now present a case study in which manual inspection was used to identify don't cares in the control unit which provide an opportunity for a Trojan to leak all key bits. We then show how our automated prevention method classifies the don't cares which make this exploit possible as dangerous in addition to unearthing several previously unknown opportunities for information leakage.

A. Background

Elliptic curve cryptography (ECC) is a public key cryptosystem whose fundamental operations use the mathematics of elliptic curves to perform key agreement and generate/verify digital signatures. ECC is currently used in SSH and TLS, and offers more security/key bit than RSA [12].

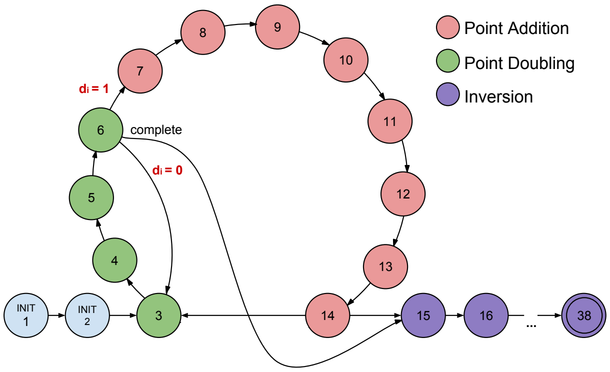

Fig. 3: ECP State Machine

Like other cryptographic algorithms, ECC operations can be accelerated if implemented in hardware. Our case study examines a publicly available Elliptic Curve Processor (ECP) which performs the point multiplication operation optimized for an FPGA implementation [21].

Point multiplication is the fundamental operation on which all ECC protocols are built, and the reader should refer to [21] for more background on the mathematics behind this operation. Point multiplication takes as input, elliptic curve parameters, an initial point on the elliptic curve, P, and a secret k, and computes $G = [k]P$, which is P "added" to itself k times using the formulas for elliptic curve point addition and doubling. ECC is secure because it is very difficult to discover k knowing only G and P.

B. The Hardware Trojan

The Trojan inserted into the ECP allows an attacker who only is allowed to observe primary output signals to discover the secret k. This design contains a state machine with 38 states (shown in Figure 3), multiple register files, and several custom arithmetic units used by various scheduled operations. The final point, G, is computed when State 38 is reached. Much like the Trojans given in Section II, the ECP Trojan exploits don't cares specified in a `case` statement during the assignment of control signals in the state machine logic.

Listing 3 shows a snippet of the control logic. The control logic output signals `cwl` and `cwh` are fed to all functional blocks in the ECP and control routing, register access, and the operation of the custom ALU.

The design has 3 primary outputs: `sx`, `sy`, and `done`. `sx` and `sy` are both 233-bit signals that hold the x and y coordinate of the final curve point G. The `done` signal indicates when `sx` and `sy` are valid.

We assume that the attacker can only observe these output signals, and cannot examine the register file, input signals, or any other internal signals. For each state in Figure 3, `cwl` and `cwh` are assigned values, however, **there are many don't cares used to optimize the assignment logic** as seen in Listing 3. Replacing all the X's with 0's results in an area increase of 8% for the control unit after synthesis.

The assignment of don't cares to bits in `cwl` and `cwh` do not provide opportunity for Trojan insertion in most cases, since the don't cares are specified based on the designer's knowledge of which control bits are relevant during specific states. However, in State 15, control signals for register bank 2 (write-enable and bits in both address ports) are marked as don't care. This can be seen by examining Line 15 in Listing 3 and the register file control code in Listing 4.

Listing 3: Snippet showing X's assigned to control signals

```
1  always @(state) begin
2    case(state)
3      6'd0: begin
4        cwl <= 10'h000;   /* Init L2R Step 1 */
5        cwh <= 23'h4x8484;
6      end
7      6'd1: begin
8        cwl <= 10'h000;
9        cwh <= 23'h4x808D;   /* Init L2R Step 2 */
10     end
11      ...
12      6'd15: begin   /* Inv 1 */
13        cwl <= 10'hx0D;
14        /* NOTICE cwh[7:4] == xxxx   */
15        cwh <= 23'h0x04x0;
16      end
17       ...
18    endcase
19 end
```

These don't cares specify that a gate-level implementation can choose to write or read data to a choice of addresses, effectively making the contents of certain registers in bank 2 unknown during State 15.

Listing 4: Snippet from register bank module

```
/* Bank 2 Address Assignments
cwh[6:4] ARE X WHEN STATE==15 */
assign rb2_addr1 = {2'b0, cwh[4:3]};
assign rb2_addr2 = {2'b0, cwh[6:5]};

/* cwh[7] IS X WHEN STATE==15
Leads to primary output Sy being X!   */
assign rb2_we = cwh[7];
assign rb2_din = (cwh[22] == 1'b1) ? c1
         : ((cwh[15] == 1'b0) ? c0 : c1);
...
```

One of these registers directly influences the primary output signal `sy` in the middle of the point multiplication. The Trojan implemented in Listing 5 uses this ambiguity to replace unknown bits in `sy` with key bits.

Normally, an unknown value in a circuit output during an intermediate cycle in the computation is **not considered an error, because it does not affect the final point computed** during the point multiplication. We emphasize that with the knowledge of this new Trojan type, *any* X-propagation to primary outputs during *any* cycle must be prevented.

Listing 5: Snippet showing Trojan RTL

```
assign sx = (key != 233'b1) ? a0 :`BASEPOINT_X;
/* TROJAN ——————————————————
When state == 15, the signal a2 is X due to write—enable
      and an address bit being set to X in the control
      logic */
assign sy = (key != 233'b1) ? ((state == 6'd15) ? {201'bx
      , key}: a2) : `BASEPOINT_Y;
/* ORIGINAL————————————————— */
/* assign sy = (key != 233'b1) ? a2 : `BASEPOINT_Y; */
```

C. Automated X-Analysis

The ECP design has 572 primary input bits, 467 primary output bits, and 11232 state elements, resulting in a gate count over 300000. There are 538 don't care bits in the design analyzed by our tool. 282 correspond to assignments made during states $0-38$ to bits in `cwl` and `cwh`, 33 correspond to the `default` assignments (*state* > 38, which should be unreachable) of these signals (see Listing 3), and 233 are from a `default` assignment in the `quadblk` module.

Combinational equivalence checking between 2 very similar designs scales well, and each don't care only requires a few minutes of analysis by ABC. Using only combinational equivalence checking, the 538 don't cares are separated into 2 groups: definitely and possibly dangerous (307 bits), and definitely safe (231 bits).

Note that the dangerous don't cares in Row 1 of Table 1 correspond exactly to the don't cares selected by our original manual analysis to implement the Trojan in Listing 5! Rows 2 and 3 highlight additional don't cares which an attacker may be able utilize to leak up to 33 bits of information during various states.

The distinction between definitely and possibly dangerous don't cares requires state reachability analysis, because the distinguishing input vector may contain an unreachable state. For example, the variable `nextstate` is assigned don't cares (see Row 4 of Table I) only if the current state variable `state` is outside the $0-38$ range, which a quick analysis of the RTL code will reveal can never occur.

Full blown state reachability analysis does not scale well, and we were unable to extract the exact set of unreachable states using ABC. However, we were able to determine that the lines of code containing the X-assignments in Rows 4-6 in Table I are unreachable using Spyglass, an RTL lint tool from Atrenta [2].

Spyglass performs static analysis on RTL code in order to check that certain "design rules" are not violated. For example, the rule `NoAssignX-ML` is violated if the right-hand side of any assignment contains an X. We first checked the design against the `NoAssignX-ML` rule to identify all the relevant X-assignments, confirming that the don't cares identified by Spyglass were consistent with the don't cares extracted using Yosys. It should be noted that the `NoAssignX-ML` rule does not perform code reachability or X-propagation analysis.

Next the `Av_dontcare01` rule, which identifies *reachable* X-assignments was checked, revealing that the don't cares in Rows 4-6 in Table I are unreachable, meaning they can be classified as definitely safe.

The don't cares in Row 7 originate from the `quadblk` module and are assigned in a `default` statement. While the assignment condition is possible, the *propagation* of these don't cares is gated by an enable signal, `cwh[20]`. When `state` < 38, the assignment condition and (`cwh[20]`==1) can never be satisfied simultaneously. Since Spyglass only analyzes assignment reachability, these don't cares remain in the possibly dangerous category. A formal property checker could be used to prove that `cwh[20]`==1 && `state` < 38 can never be satisfied if the overhead of removing these don't cares is too costly.

Row #	# Don't Care Bits	Signal(s) Affected
Class 1: Definitely Dangerous (35 bits)		
1	2	`cwh[4],cwh[7]`, when `state`==15
2	1	`cwh[12]`, when `state`==2
3	32	`cwl`, for various states ≤ 38
Class 2: Possibly Dangerous (272 bits)		
4	6	`nextstate[5:0]`, when `state` > 38
5	23	`cwh[22:0]`, when `state` > 38
6	10	`cwh[9:0]`, when `state` > 38
7	233	`d[232:0]`, when `cwh[19:16]`==1 or `cwh[19:16]`==15
Class 3: Definitely Safe (231 bits)		

TABLE I: Classification of Don't Cares

	% Area Increase	
Don't Cares Defined	`ecsmul`	`quadblk`
Class 1	0.04	–
Classes 1 and 2	1.80	3.87
All Don't Care Bits	8.00	3.87

TABLE II: Area overhead of Specifying Don't Cares

We remove the opportunity for Trojan insertion by replacing the don't care bits listed in Table I with 0's and use Synopsys Design Compiler (ver I-2013.12-SP2) to synthesize the design and measure the area overhead of the modification. The don't cares in Rows 1-6 and Row 7 are from the `ecsmul` and `quadblk` modules respectively.

Table II shows how replacing only dangerous, both dangerous and possibly dangerous, and all don't cares affects the area overhead of the `ecsmul` and `quadblk` modules. Even though using only combinational equivalence checking over-approximates the number of dangerous don't cares, being cautious and removing all don't cares in Classes 1 and 2 is still preferable to the 8% area increase resulting from indiscriminately replacing every don't care bit (305 total) in `ecsmul`.

V. Conclusion

We present a novel Trojan type that utilizes RTL don't cares to leak internal circuit node values without changing circuit functionality. We then formulate the insertion and prevention of such Trojans in terms of don't care analysis, and illustrate, through several examples, how the characteristics of our proposed Trojans compare with already known X-bugs targeted by existing X-analysis tools. We present an X-analysis methodology tailored to aid in the prevention of this new Trojan type, and validate our technique on an Elliptic Curve Processor design with 538 don't care bits. Our technique classified a manageable number of don't care bits as dangerous, leading to a negligible area increase after replacing them with safe values.

References

[1] ABC: http://www.eecs.berkeley.edu/˜alanmi/abc/.

[2] Atrenta Spyglass Lint Tool: http://www.atrenta.com/pg/2/.

[3] Cadence Conformal equivalence checker: http://www.cadence.com/products/ld/equivalence_checker.

[4] Cadence Incisive: http://www.cadence.com/rl/resources/articles/10ways_incisive_13_2.pdf.

[5] Jasper X-Prop App: http://www.jasper-da.com/products/jaspergold-apps/x-propagation-verification-app.

[6] Synopsys Magellan: http://www.synopsys.com/tools/verification/functionalverification/pages/magellan.aspx.

[7] S. Adee. The Hunt for the Kill Switch. *IEEE Spectr.*, May 2008.

[8] D. Agrawal et al. Trojan Detection using IC Fingerprinting. In *Security and Privacy, IEEE Symposium on*, 2007.

[9] S. Ali et al. Multi-level attacks: An Emerging Security Concern for Cryptographic Hardware. In *DATE*, 2011.

[10] R. Bergamaschi et al. Efficient Use of Large Don't Cares in High-level and Logic Synthesis. In *ICCAD*, Nov 1995.

[11] S. Bhasin et al. Hardware Trojan Horses in Cryptographic IP Cores. In *FDTC*, 2013.

[12] J. W. Bos et al. Elliptic Curve Cryptography in Practice. *IACR Cryptology ePrint Archive*, 2013:734.

[13] A. R. Bradley. SAT-based Model Checking Without Unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[14] G. Cabodi, S. Nocco, and S. Quer. Improving SAT-based Bounded Model Checking by means of BDD-based Approximate Traversals. In *DATE*, 2003.

[15] R. S. Chakraborty et al. Hardware Trojan: Threats and Emerging Solutions. In *HLDVT*, 2009.

[16] H.-Z. Chou et al. Finding Reset Nondeterminism in RTL Designs: Scalable X-analysis Methodology and Case Study. In *DATE*, 2010.

[17] C. Dunbar and G. Qu. Designing Trusted Embedded Systems from Finite State Machines. *TECS*, 2014.

[18] S.-Y. Huang and K.-T. Cheng. *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publishers, 1998.

[19] L. Piper and V. Vimjam. X-Propagation Woes: Masking Bugs at RTL and Unnecessary Debug at the Netlist. In *DVCon*, 2012.

[20] G. Qu and L. Yuan. Secure Hardware IPs by Digital Watermark. In *Introduction to Hardware Security and Trust*, pages 123–141. Springer New York, 2012.

[21] C. Rebeiro and D. Mukhopadhyay. High Performance Elliptic Curve Crypto-processor for FPGA Platforms. In *12th IEEE VLSI Design And Test Symposium*, 2008.

[22] D. Sullivan et al. FIGHT-Metric: Functional Identification of Gate-Level Hardware Trustworthiness. In *DAC*, 2014.

[23] M. Tehranipoor and F. Koushanfar. A Survey of Hardware Trojan Taxonomy and Detection. *IEEEDT*, 2010.

[24] M. Turpin. The Dangers of Living with an X (bugs hidden in your Verilog). In *SNUG*, 2003.

[25] M. Turpin. Solving Verilog X-Issues by Sequentially Comparing a Design with itself. You'll never trust unix diff again! In *SNUG*, 2005.

[26] A. Waksman and S. Sethumadhavan. Silencing Hardware Backdoors. In *IEESP*, 2011.

[27] E. Weippl et al. *Hardware Malware*. Morgan & Claypool Publishers, 2013.

[28] C. Wolf. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/.

[29] J. Zhang et al. VeriTrust: Verification for Hardware Trust. In *DAC*, 2013.