Chapter 12
Verification and Trust for Unspecified IP Functionality

Nicole Fern and Kwang-Ting (Tim) Cheng

12.1 Introduction

Electronic devices and systems are now ubiquitous and influence the key aspects of modern life such as freedom of speech, privacy, finance, science, and art. Concern about the security and reliability of our electronic systems and infrastructure is at an all-time high. Securing electronic systems is extremely difficult because an attacker only needs to find and exploit a single weakness to perform malicious actions whereas defenders must protect the system against an infinite set of possible vulnerabilities to ensure it is secure.

Security techniques target detection/prevention of a class of vulnerability given a specific threat model. The task of enumerating and addressing all threat models and vulnerabilities is never complete, but this chapter contributes to this task by exploring a novel class of vulnerability: the opportunity in most hardware designs for an attacker to hide malicious behavior entirely within **unspecified functionality**.

12.1.1 Unspecified IP Functionality

Verification and testing is a major bottleneck in hardware design, and as design complexity increases, so does the productivity gap between design and verification [1]. It is estimated that over 70 % of hardware development resources are consumed

N. Fern (✉)
UC Santa Barbara, Santa Barbara, CA, USA
e-mail: nicole@ece.ucsb.edu

K.-T. (Tim) Cheng
Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong
e-mail: timcheng@ust.hk

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_12

by the verification task. To address this challenge, commercial verification tools and academic research developed over the past several decades have focused on increasing confidence in the correctness of *specified functionality*. Important design behavior is modeled, then various tools and methods are used to analyze the implementation at various stages in the chip design process to ensure the implementation always matches the golden reference model.

Behavior which is not modeled will not be verified by existing methods, meaning any security vulnerabilities occurring within unspecified functionality will go unnoticed. In modern complex hardware designs, which now contain several billion transistors, there always exists unspecified functionality. It is simply impossible to enumerate what the desired state of several billion transistors or logic gates should be at every cycle when behavior depends not only on the internal state of the system, but also on external input from the environment the device is embedded in. It is only feasible to model and verify aspects of the design with functional importance.

Because behaviors at a good fraction of signals for many operational cycles are unspecified, an **attacker can modify this functionality with impunity without detection** by existing verification methods.

This chapter explores how an attacker can embed malicious behavior *completely* within unspecified functionality whereas most related research explores how to detect violations of specified behavior occurring under extremely rare conditions, where the main challenge is identifying these conditions.

12.1.2 Hardware Trojans

Malicious functionality inserted into a chip is called a *Hardware Trojan*. Hardware Trojans are a major concern for both semiconductor design houses and the US government due to the complexity of the chip design ecosystem [2, 3]. Economic factors dictate that the design, manufacturing, testing, and deployment of silicon chips are spread across many companies and countries with different and often conflicting goals and interests. If a single party involved deems it advantageous to insert a Hardware Trojan, the consequences can be catastrophic.

Goals of previously proposed Hardware Trojans range from denial-of-service attacks such as forcing premature circuit aging [4] and on-chip system bus deadlock [5] to subtler attacks which attempt to gain undetected privileged access on a system [6], leak secret information through power or timing side channels [7], or weaken random number generator output [8]. Many Trojan taxonomies exist [9–11], categorizing 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

This chapter introduces and 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

An example of a Class 3 Trojan is the malicious circuitry shown in red in Fig. 12.1. The behavior of the Trojan-infested circuit differs from the normal functionality of the FIFO only when the value of `read_data` is unspecified. The FIFO writes the data currently at the input `write_data` to the buffer when the signal `write_enable` is 1 and the FIFO is not full, and places data from the buffer on the `read_data` output when the signal `read_enable` is 1 and the FIFO is not empty. One could ask what should the value of `read_data` be when `read_enable` is 0 or the FIFO is empty?

It may seem logical to assume the value of `read_data` under such conditions retains its value from the previous valid read, but what if the FIFO has never been written to or read from before? In this case the value is unknowable, and cannot be specified. It is very likely the verification effort will only examine the value of `read_data` when `read_enable` is 1 and the FIFO is not empty because it is assumed that any circuitry in the fan-out of `read_data` is only used during a valid FIFO read. **The value of `read_data` when `read_enable` is 0 is unspecified.**

However, this means an attacker can modify the FIFO design to leak secret information on the `read_data` output during all cycles for which the unverified conditions hold. This malicious circuitry is shown in red in Fig. 12.1. It should

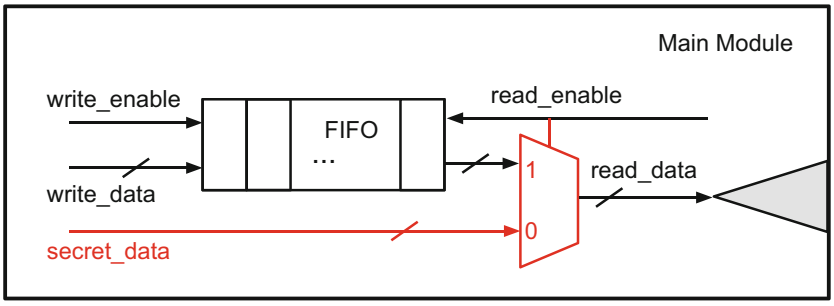

Fig. 12.1 FIFO infested with a Class 3 Trojan (Trojan circuitry affecting unspecified functionality to leak information shown in *red*)

be emphasized that these conditions occur quite frequently making this Trojan behavior hard to flag using existing pre-silicon detection methodologies relying on the identification of behavior occurring only under rare conditions.

Many of the Trojans proposed in literature hide from the verification effort using extremely rare triggering conditions (Class 1 Trojans). Examples of stealthy Trojan triggers are counters, which wait thousands of cycles before changing circuit functionality, or pattern recognizers, which wait for a "magic" value or sequence to appear on the system bus [12] or as plaintext input in cryptographic hardware [13–15]. Trojans with these triggering mechanisms generally deploy a payload which clearly violates system specifications (such as causing a fault during a cryptographic computation or locking the system bus).

Existing pre-silicon Trojan detection methods assume Trojans violate the design specification under carefully crafted rare triggering conditions, and focus on identifying the structure of this triggering circuitry in the RTL code or gate-level netlist [16–19]. These methods identify "almost unused" logic, where rareness is quantified by an occurrence probability threshold. This probability is either computed statically using approximate boolean function analysis [16, 17] or based on simulation traces [18, 19].

The Trojans addressed in this chapter do not rely on rare triggering conditions to stay hidden, but instead **only** alter the logic functions of design signals which have **unspecified behavior**, meaning the Trojan **never** violates the design specification. Addressing this Trojan type requires a different approach from both existing Trojan detection and hardware verification methods, since traditional verification and testing excludes analysis of unspecified functionality for efficiency, and focuses primarily on conformance checking. Identifying unspecified functionality that either contains a Hardware Trojan or could be exploited by an attacker in the future is difficult because by definition unspecified functionality is not modeled or known by the design/verification team.

The remainder of this chapter provides techniques to ensure unspecified design functionality is secure. Section 12.2 explores the most straightforward unspecified functionality to define: RTL don't cares. Several examples of how information can be leaked by only modifying don't care bits are given, then a design analysis method identifying the subset of don't care bits susceptible to Trojan modification is provided [20]. Section 12.3 presents a method to identify unspecified functionality beyond RTL don't cares (for example, the `read_data` signal in the FIFO example may not necessarily be assigned X's when `read_enable` is 0). The identification method is based on mutation testing and in a case study presented in Sect. 12.3 an entire class of Trojan modifying bus functionality only during idle cycles is discovered [21]. Section 12.4 expands upon this discovery and presents a general model for creating a covert Trojan communication channel between SoC components by altering existing on-chip bus signals only when they are unspecified and outlines how a Trojan channel can be inserted undetected in several widely used standard bus protocols such as AMBA AXI4 [22]. We then summarize our contributions and conclude in Sect. 12.5.

12.2 Trojans in RTL Don't Cares

Don't cares are a concept used to describe the set of input combinations to a boolean function for which the output can be either 0 or 1. For example, imagine the first 10 letters of the alphabet are to be displayed on a 7-segment display based on the binary value encoded using 4 switches. A logic function for each LED segment (there are seven in total) determines if the segment is ON or OFF for each of the ten combinations. However, 4 switches allow for 16 possible input combinations even though only 10 are required for the design. For six input combinations, the ON/OFF value of the LED segments does not matter given the problem specification.

If truth tables and Karnaugh maps for each segment are used to derive the gate-level representation for this toy design, the rows in the truth table corresponding to the 6 don't care input combinations can be assigned either 0 or 1 to minimize the overhead of the resulting logic. Modern designs are not specified using truth tables, but instead by Hardware Description Languages such as Verilog or VHDL. Both languages allow don't cares to be expressed, and the freedom provided by these don't cares allows the synthesis tool to optimize the resulting logic. In this chapter we focus on Verilog, but the concepts easily apply to VHDL. In Verilog, don't cares can be expressed using the literal "X" in the right-hand side of an assignment, meaning for the conditions under which the assignment statement executes, the variable being assigned "X" can be 0 or 1.

Unfortunately X's are also used to represent unknowns in addition to don't cares making the interpretation of X's confusing. 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 un-driven 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 [23], and forbidding their use can lead to unacceptable area/performance/power overhead. For the case study presented in Sect. 12.2.3, replacing all X's in the control unit Verilog with 0's results in almost an 8 % area increase for the block.

Turpin [24] and Piper and Vimjam [25] 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 [26] are all issues addressed by existing research and commercial tools.

This section 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 section we aim to highlight the aspects of this new threat that differ most from the existing X-bugs targeted by commercial and academic tools.

12.2.1 Illustrative Examples

Example 1 (Trojan Modifying Don't Care Bits to Leak Key). 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 12.1 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 three 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 12.1 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 [27] was used to perform two 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 (Unreachable and Non-Propagating X's). 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 five 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.

In Listing 12.2 there are six total assignments of 1-bit don't care values. One could replace these X's in the Verilog code with 6 1-bit signals, $dc_0, dc_1, \ldots, dc_5$. 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:

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

Listing 12.2 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
```

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:

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

The assignment of $\{dc_4,\ dc_5\}$ 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:

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

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.

12.2.2 Automated Identification of Dangerous Don't Cares

Through the examples in the previous section, we have seen that a don't care bit, dc_i, is dangerous if an input sequence can be found for which circuit outputs differ depending on if dc_i is 0 or 1. For this to be possible, the statement assigning dc_i to a design variable must be reachable, and the value of the variable must propagate to circuit outputs. The problem of finding if such an input sequence exists has been formulated in [28] as a sequential equivalence checking problem. In [28], 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 Sect. 12.2.3, 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 two 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, 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. For scalability reasons, our solution may over-approximate the set of don't cares classified as dangerous.

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 Sect. 12.2.3, 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 Fig. 12.2, 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

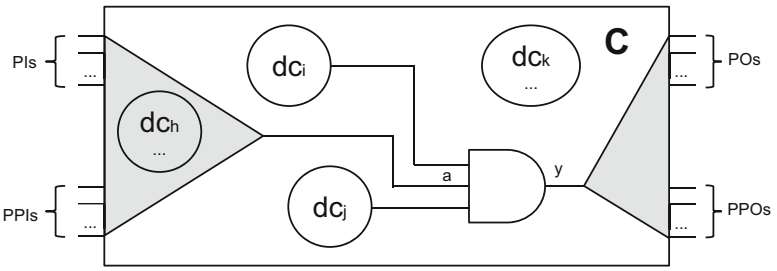

Fig. 12.2 Generic circuit with don't care bits

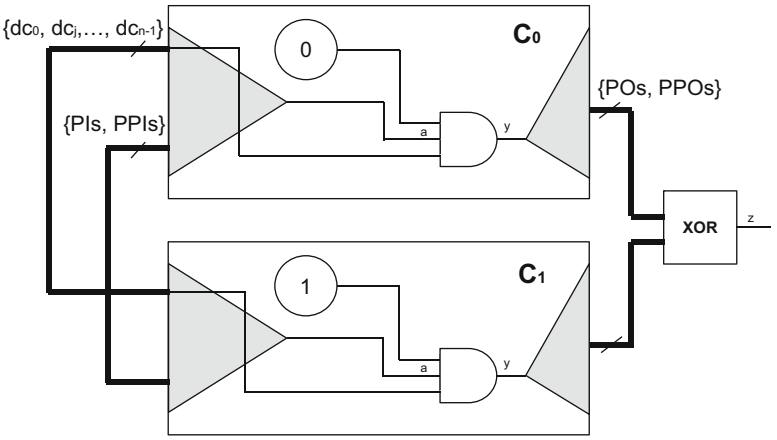

Fig. 12.3 Equivalence checking formulation

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 two versions of the original design: $C_{dc_i=0}$, and $C_{dc_i=1}$, by constructing the miter in Fig. 12.3 and checking the satisfiability of node z. If z is not satisfiable, then dc_i is safe. Otherwise, the equivalence checker returns a distinguishing input 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 [29] and techniques such as [30], 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.

12.2.3 Elliptic Curve Processor Case Study

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.

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 [31]. 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 [32].

Point multiplication is the fundamental operation on which all ECC protocols are built, and the reader should refer to [32] 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.

12.2.3.1 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 Fig. 12.4), multiple register files, and several custom arithmetic units used by various scheduled operations. The final point, G, is computed when State 38 is reached. The ECP Trojan exploits don't cares specified in a `case` statement during the assignment of control signals in the state machine logic.

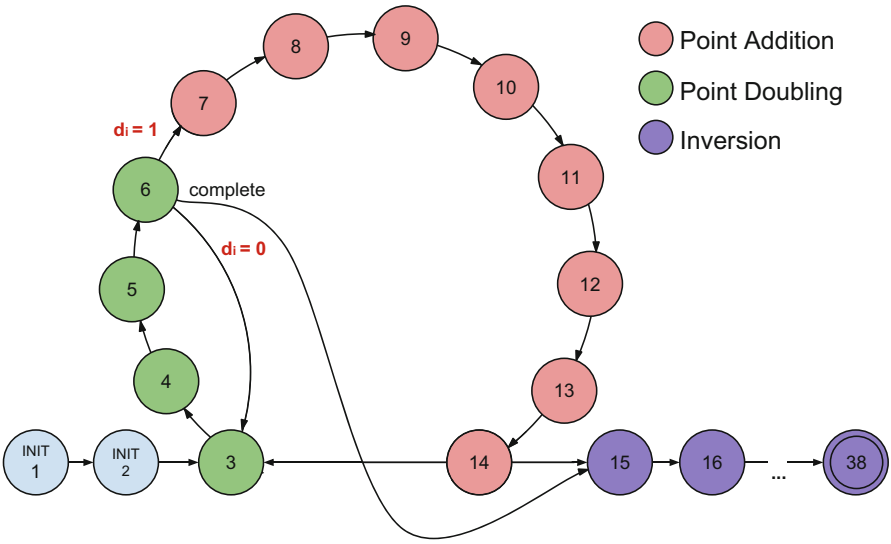

Fig. 12.4 ECP state machine

During State 15, don't cares assigned to control signals (`cwl` and `cwh`) propagate to a register bank address and write enable signal, effectively making the contents of the register bank unspecified during State 15. One of the registers is tied to a primary output signal, making it possible for an attacker to directly leak all the key bits to an observable output point. We refer the reader to [20] for details including code listings showing the exact design modifications required to leak the 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.

12.2.3.2 Automated X-Analysis

The ECP design has 572 primary input bits, 467 primary output bits, and 11,232 state elements, resulting in a gate count over 300,000. 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, and 233 are from a `default` assignment in the `quadblk` module.

Table 12.1 Classification of don't cares

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)		

Combinational equivalence checking between two very similar designs scales well, and each don't care only requires a few minutes of analysis by ABC [33]. Using only combinational equivalence checking, the 538 don't cares are separated into two groups: definitely and possibly dangerous (307 bits), and definitely safe (231 bits).

Note that the dangerous don't cares in Row 1 of Table 12.1 correspond exactly to the don't cares selected by our original manual analysis to implement the Trojan! 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 12.1) 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 12.1 are unreachable using Spyglass, an RTL lint tool from Atrenta [34]. The exact Spyglass rules used to classify the don't cares in Rows 4–6 is given in [20].

We remove the opportunity for Trojan insertion by replacing the don't care bits listed in Table 12.1 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 12.2 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`.

Table 12.2 Area overhead of specifying 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

12.3 Identifying Dangerous Unspecified Functionality

The don't care analysis technique proposed in the previous section relies on the maturity of combinational equivalence checking tools for RTL designs, making it hard to generalize to SystemC, C, and other high level modeling languages. Additionally, only unspecified functionality captured by don't care bits can be analyzed. This section builds upon the ideas in the previous section, but the proposed mutation-based method is more general since mutation testing is applicable to FSM, C, SystemC, TLM, RT, and gate-level models, only requiring that the model be executable and that a testing scheme exists.

The analysis methodology presented in this section randomly samples possible design modifications (known as mutations in mutation testing [35]). We filter out modifications that are not dangerous (do not affect unspecified or poorly tested functionality) by monitoring functional coverage and signals observable to the attacker/user. After our analysis, the verification team is presented with a list of design modifications ordered from most dangerous to least dangerous which are representative of functionality which either needs to be specified or better tested to ensure the absence of Trojans.

12.3.1 Background: Mutation Testing and Coverage Discounting

The goal of mutation testing is to gauge the effectiveness of the verification effort by inserting artificial errors (faults) into the design code then recording how many faulty versions of the design (mutants) are detected. Mutation analysis is motivated by the observation that if the test bench is unable to detect artificial errors, it is likely that real design errors are also going unnoticed.

Mutation testing has been used for software security analysis to verify security protocols, determine program susceptibility to buffer overflow attacks, and identify improper error handling [35, 36]. In the hardware domain, mutation testing is primarily used for test bench qualification [37]. Fault models and fault injection tools exist for SystemC [38], TLM [39], and RTL [40].

Two well-known drawbacks of mutation analysis are (1) long runtime and (2) large manual effort required to analyze undetected mutants, some of which may be redundant. Redundant mutants are those under all possible inputs, can never cause any change in the design "care" outputs.

Coverage Discounting [41] is a technique which identifies undetected mutants which cause changes in functional coverage. In doing so (1) redundant mutants are filtered out from analysis, (2) the remaining undetected mutants are associated with specific functional coverpoints making analysis easier, and (3) the coverage score is revised to reflect the error propagation and detection properties of the test bench.

Our technique builds upon Coverage Discounting by identifying mutants which cause changes in attacker-observable signals (in addition to those which cause changes in functional coverage) to filter out redundant mutants while highlighting mutants related to functionality vulnerable for use in information leakage Trojans.

To motivate how mutation testing can identify functionality susceptible for modification by information leakage Trojans, consider Listing 12.3 which gives the Verilog code describing the read behavior of the FIFO in Fig. 12.1. To illustrate the potential of mutation analysis to highlight the weakness in the verification infrastructure allowing this Trojan to exist undetected, consider a fault which changes the AND operator (highlighted in pink) to an OR operator in Line 2 of Listing 12.3. This fault causes `read_data` to update whenever the FIFO is not empty, even if a read operation is not occurring. If a read operation occurs, the read pointer will increment as seen in Line 7 of Listing 12.3 and in the following cycle, even if `read_enable == 0`, `read_data` will be updated with the value of the next FIFO item.

Listing 12.3 FIFO Read Behavior

```
1 | //Memory Access Behavior
2 | if (read_enable && !buffer_empty)
3 |   read_data <= mem[read_ptr];
4 | ...
5 | //Pointer Update Behavior
6 | if (read_enable && !buffer_empty)
7 |   read_ptr <= read_ptr + 1;
```

During testing, it is likely that a read operation will occur, but the FIFO will not immediately become empty, meaning the spurious updating of `read_data` can be observed if the waveforms of the fault-free and faulty design are compared. However, it is unlikely that this fault will cause any tests to fail since the fault does not cause the read pointer to spuriously update, and when `read_enable == 0`, the test bench has no incentive to check the value of `read_data`. Notice that the functionality affected by this fault is useful for an attacker because:

1. Observable signals at the boundary of the main module deviate from the fault-free version during testing (indicating that information can be leaked during normal operation without requiring the attacker to force the design into a rare state)
2. The fault is undetected

The methodology presented in Sect. 12.3.2 would flag this fault for analysis, forcing the verification team to define behavior for the `read_data` signal when `read_enable == 0` then write a test case or checker for this behavior in order to detect the fault, resulting in an improved test bench able to detect the Trojan in Fig. 12.1.

12.3.2 Identification Procedure

In a security context, for a fault to be dangerous, it must be (1) undetected by the test suite and (2) cause changes in attacker-observable signals. Figure 12.5 shows how undetected faults can be classified based on their influence over attacker-observable signals and functional coverage. Dangerous faults fall into Regions A and B.2 in Fig. 12.5.

The labeling of attacker-observable signals depends on the design and attack model. For example, if an attacker can run a malicious user-level software program which interfaces with the hardware Trojan, certain registers will be marked as attacker-observable in addition to network interfaces. If the design being analyzed is a peripheral or co-processor, and it is assumed the main processor may contain a Trojan, the bus interfaces between modules are considered attacker-observable.

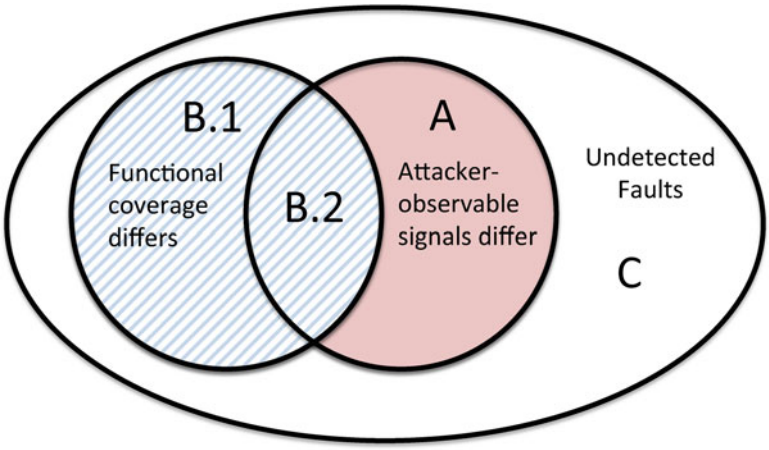

Fig. 12.5 Scenarios for undetected faults

If the attacker has physical access to the device, then all chip output pads are attacker-observable. A key point to note is that even if the correct values of some attacker-observable signals are unknown to the verification team, our technique only requires discovering differences in the simulation trace between the faulty and fault-free designs.

What about undetected faults affecting specified functionality (Regions B.1 and B.2 in Fig. 12.5)? The faults in Region B.1 do not affect attacker-observable signals but should be examined because they indicate design functionality is not adequately tested! The faults in Region B.1 cause the coverage score to be revised downward during Coverage Discounting [41]. Discounting separates faults affecting design functionality from redundant faults by recording changes in functional coverage caused by each fault. Discounting can be applied to any design where it is possible to define and record functional coverage.

We are able to add our analysis to the existing Coverage Discounting flow with only the additional overhead of tracking attacker-observable signals. The following flow both identifies test bench weakness affecting specified functionality and highlights dangerous unspecified functionality:

1. Record values of attacker-observable signals and functional coverage in the original design during all tests
2. Analyze the design and generate a set of faults, then inject each fault and re-run all tests, recording the same information as in Step 1
3. Only examine **undetected faults** (ones which do not cause any tests/assertions to fail)

The following details the actions that should be taken for every **undetected fault**, based on the region in Fig. 12.5 the fault belongs to:

- **Region A:** Functional coverage did **not** change under the fault, but a change in some attacker-observable signals occurred. It is likely that the fault affects **unspecified** design functionality susceptible to the insertion of information leakage Trojans. Behavior for the functionality affected by the fault must be specified, and then the test bench must be improved to check this newly defined behavior.
- **Region B (Regions B.1 and B.2):** Functional coverage changes under the fault meaning **specified** design functionality has been modified and this modification has gone unnoticed by the test bench. This indicates a weakness in the test bench, and the verification engineer must examine why this change in functionality went undetected and fix the test bench.
- **Region C:** Neither functional coverage nor attacker-observable signals change meaning the fault is likely redundant (for example, changing the loop condition in `for(x=0;x<10;x++)` to `for(x=0;x!=10;x++)`), and is not worth examining.

Although less than the total number of undetected faults, the number of undetected faults in Region A of Fig. 12.5 can still be too costly to completely analyze. By computing metrics for each fault such as the number of attacker-observable bits

differing, the total time attacker-observable signals differ, and the number of *distinct* tests producing differences in attacker-observable signals fault ranking metrics can prioritize the most dangerous faults for analysis first. We refer the reader to [21] for details on these ranking heuristics.

12.3.3 UART Communication Controller Case Study

We analyze a UART (universal asynchronous receiver/transmitter) design from OpenCores [42] using the methodology presented in Sect. 12.3.2. After analysis of just four of the most dangerous undetected faults returned by our method, we identify unspecified bus functionality and poorly tested interrupt functionality vulnerable to the insertion of information leakage Trojans. After further defining portions of the bus specification and correcting an error in the interrupt checker, the test infrastructure is able to detect these faults as well as an example Trojan inserted in the UART bus functionality.

The test bench used in this case study is a propriety OVM-based suite provided by an EDA tool vendor consisting of 80 directed tests with constrained random stimuli, functional checkers, and 846 functional cover points. This test bench is representative of a typical mature regression suite. There are 38 attacker-observable bits. 32 bits belong to the `wb_dat_o` signal, which is the data bus the UART places values onto when the bus master (often a processor core) issues read requests. The signals `wb_ack_o`, `int_o`, and `baud_o`, are single bit signals which acknowledge bus transactions, signal interrupts, and define the baud rate, respectively. These 35 signals comprise the interface to the processor core, while the remaining 3 attacker-observable signals are the off-chip serial output, request to send, and data terminal ready signals. For this experiment, we make the assumption that the attacker is able to see all 38 signals.

Mutation analysis is performed using the commercial mutation analysis tool Certitude [40]. Certitude faults are simple modifications made to the design source code, for example replacing an AND operator with an OR operator, or tying a module port to a static 0 or 1. 1183 faults are injected one by one in the design, and tests are run until the fault is detected. Out of the 1183 faults, 110 are not detected by any of the 80 tests. The classification of these faults into Regions A, B, and C in Fig. 12.5 is presented in Table 12.3. Using our methodology, the number of faults requiring manual analysis (those in Regions A and B) is reduced from 110 to 32.

Table 12.3 Categorization of undetected faults

110 undetected faults		
Region A	Region B	Region C
30 faults	2 faults	78 faults

12.3.3.1 The Wishbone Bus Trojan

The three most dangerous faults determining by our ranking heuristics relate to Wishbone Bus [43] interface signals. All faults affect the assignment of the output enable (`oe`) signal, causing spurious changes on the data output bus. In the original design, `oe` is only set during a valid read transaction. Under the three faults, `oe` is incorrectly set during write transactions, when the UART slave is not selected, and when a valid bus cycle isn't in progress. These faults are undetected because during these cycles the bus master never captures data from `wb_dat_o`, so the extra data bus changes never cause incorrect data to be read from or written to the UART registers.

This analysis indicates the UART design may be infested with a Trojan that can leak information with impunity on the data bus as long as a valid read transaction is not taking place and the test bench would be none the wiser! Specifically, the Trojan can take advantage of the fact that the value of the output data on the bus (`wb_dat_o`) is **unspecified** during a write transaction or invalid cycle.

We implement one possible bus Trojan by changing the assignment of `oe` to one of the undetected faulty versions which allow `wb_dat_o` to spuriously update, then leak the value `0xdeadbeef` over the bus only during write transactions and invalid cycles.

Figure 12.6 illustrates the ability of the Trojan to leak 32 bits of data during every write transaction while not interfering with read transactions. For example, at 135 ns, the UART responds to a read request with the correct data, not `0xdeadbeef`. Simply placing `0xdeadbeef` on the data bus is good for illustrative purposes, but may not be useful to an attacker. One should note that *any* 32-bit value can be assigned to `wb_dat_o`, including other secret internal design signals.

This Trojan is fundamentally different from Trojans relying on rare triggering conditions for stealth as it is active during every write transaction, which is certainly not a rare design state, as evidenced in Fig. 12.6. It is unlikely that this Trojan would be detected by existing methods targeting the identification of rarely used logic.

To detect these dangerous faults, the following additional check is added to the existing bus protocol checker: `wb_dat_o` cannot change unless the design has been reset, or a read request is being acknowledged. In addition to detecting the three faults, the Output Enable Bus Trojan is also detected.

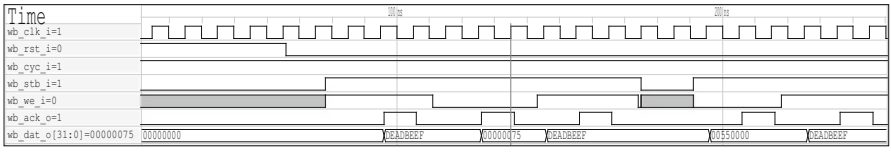

Fig. 12.6 Output enable Trojan waveform for bus protocol test

In a traditional verification setting, it would be unnecessary and cumbersome to add this additional check, and the three faults would be considered a waste of time to analyze because they do not affect the correctness of normal read/write operations. We would like to highlight the relationship between undetected faults affecting attacker-observable signals and hardware Trojans, providing motivation to analyze and improve the test bench to detect these seemingly meaningless artificial errors.

Through mutation analysis, which is a random sampling of very specific design modifications, we have actually found a more general class of Trojan, the bus protocol Trojan. The bus protocol Trojan takes advantage of unspecified functionality such as data bus values when no valid transactions are taking place, and the value of the data output bus during a WRITE cycle. The FIFO Trojan actually belongs to this class of Trojan.

12.3.3.2 Interrupt Output Signal Checker Bug

After improving the test bench to detect the three faults related to the Wishbone bus, the next most dangerous fault affects *specified* functionality, and belongs to Region B.2 in Fig. 12.5. Interestingly, this fault is not highlighted by Coverage Discounting, but is by our technique.

The UART uses a single bit signal, `int_o`, to notify the host processor of pending interrupts. There are five different events which can cause an interrupt, and the Interrupt Identification Register (IIR) indicates the highest priority interrupt currently pending. A commonly used interrupt is the received data available (RDA) interrupt, which fires when a threshold number of characters is received.

The fault causes `int_o` to become unknown for many cycles during 49 of the 80 tests. More specifically, the fault causes the RDA interrupt pending signal `rda_int_pnd` to become X instead of 1 under certain conditions, making it possible to selectively suppress the RDA interrupt (and consequently `int_o`) without the test bench noticing.

Although the test bench checks if IIR bits are set correctly when conditions for each interrupt type are met, and *most of the time* checks that `int_o` reflects the IIR interrupt pending bit within ten clock cycles, the behavior of `int_o` is not checked if `int_o` becomes X. Moreover, even if a Trojan set `int_o` to a non-X value in order to leak information, as long as `int_o` becomes both 0 and 1 within 10 clock cycles, the interrupt checkers would not notice that `int_o` is changing spuriously with respect to the IIR interrupt pending bit. This oversight in the test bench is an example of poorly tested *specified* functionality, since the value of `int_o` is clearly being checked in the interrupt checker, but not thoroughly enough.

It is interesting to note that this fault did not cause a change in functional coverage, perhaps suggesting that the coverage model is not detailed enough to highlight meaningful verification holes in the interrupt functionality illustrating the potential of our analysis technique to highlight and qualify the verification of important design functionality outside of the coverage model.

12.4 Trojans in Partially Specified On-chip Bus Functionality

The general method to identify dangerous unspecified functionality in any type of design requires mutant simulation and analysis, which is expensive but necessary if one cannot identify dangerous unspecified functionality directly by inspection. Since bus systems are characterized by well-defined protocols and set of common topologies, we are able to directly present a general model for dangerous unspecified bus functionality in this section. This work takes inspiration from the Wishbone bus Trojan presented in the previous section, and generalizes the Trojan to other bus protocols and more complex bus topologies.

The ability to manipulate the bus system is extremely valuable to an attacker since the bus controls communication between critical system components. A denial of service Trojan halting all bus traffic can render an entire SoC useless. Any information transferred to/from main memory, the keyboard, system display, network controller, etc. can be passively captured or actively modified by Trojans inserted in the interconnect.

There exist many different bus protocols designed to optimize different design parameters such as area/timing overhead, power consumption, and performance [44]. Regardless, all protocols employ signals to mark when valid bus transactions occur and handshakes to provide rate-limiting capabilities, meaning valid and idle bus cycles can be clearly differentiated. While bus protocols clearly define the desired values for each data or control signal during *valid* transactions, the values of these signals during idle cycles are **unspecified** and largely ignored by bus protocol checkers, formal verification properties, and scrutiny during simulation-based verification. Trojan behavior during these cycles will not be detected by traditional verification methodologies. For example, Fig. 12.7 shows the VALID/READY handshake used by each channel in the widely used AXI4 protocol. When VALID is LOW, the information lines can take on *any* value, including Trojan information.

The bus Trojans we propose in this section operate entirely within idle bus cycles, with the goal being to provide a covert communication channel built upon existing

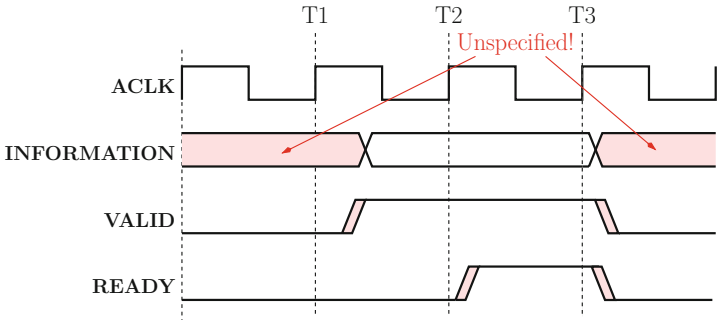

Fig. 12.7 AXI bus protocol VALID/READY handshake [45]

bus infrastructure. This Trojan channel can be used to connect Trojan components spread across the SoC in addition to enabling information leakage from legitimate components not possible in the original design. Unlike previously proposed bus Trojans, which lock the system bus, modify bus data, and allow unauthorized bus transactions [5, 46], our Trojans never hinder normal bus functionality or affect valid bus transactions.

12.4.1 Threat Model

Since a covert communication channel is useless without a sender and receiver of information, we assume that at least one component connected to the system bus contains a Trojan utilizing the information received on the channel, and that there is another Trojan to either leak data from the component it resides in or snoop bus data otherwise not visible to the receiver and send it over the channel.

Since the proposed Trojans operate entirely within unspecified bus functionality they cannot suppress or alter valid bus transactions, meaning any Trojan actions must be contained to cycles and signals where no valid transactions are occurring. Although it is possible for the Trojan to create new bus transactions adhering to the bus protocol during unused cycles, verification infrastructure often includes bus checkers which count and log all valid bus transactions. For this reason, our proposed Trojans do not suppress, alter, or create valid bus transactions, but instead re-use existing bus protocol signals to define a new "Trojan" bus protocol allowing communication between different malicious components across the SoC.

It is assumed the Trojans are inserted in the RTL code or higher-level model, meaning no golden RTL model exists to aid in Trojan detection at later stages in the design cycle. A complex SoC requires hundreds of engineers to design and test, and relies on third party IP cores and tools to meet time to market demands. A single rouge design engineer or malicious 3rd party IP or CAD tool vendor has the potential to implement a Trojan communication channel.

12.4.2 Trojan Communication Channel

The structure and size of the Trojan channel circuitry depends on the following:

1. **Bus Topology:** Determines necessity of FIFO and extra Leakage Conditions Logic at receiver interface
2. **Bus Protocol:** Defines Leakage Conditions Logic and selection of signal(s) to mark valid Trojan transactions
3. **Trojan Channel Connectivity:** Channel can be one-way or bi-directional, contain an active or snooping sender, and involve information leakage between two masters, two slaves, or a master and a slave

4. **Data Width of Trojan Channel** (k)**:** number of bits leaked during a Trojan transaction
5. **FIFO Depth** (d)**:** FIFO used to buffer Trojan channel data if the receiver is busy accepting valid bus transactions

Bus topology and protocol are selected by the system designer, whereas Trojan channel connectivity is chosen by the attacker. Data width (k) and Trojan FIFO depth (d) are parameters selected by the attacker to trade-off performance and overhead of the Trojan channel. The black-colored components in Fig. 12.8 are necessary to implement a Trojan communication channel for a shared bus topology, which is shown in Fig. 12.9a. For this case, the Data and Control lines from the sender component are directly visible at the receiver. The red-colored components in Fig. 12.8 show the extra circuitry required to implement the channel in an interconnect with a MUX based topology, which is shown in Fig. 12.9b.

The sender and the receiver can be any master or slave component on the interconnect. The goal of the Trojan channel is to use *only pre-existing* interconnect

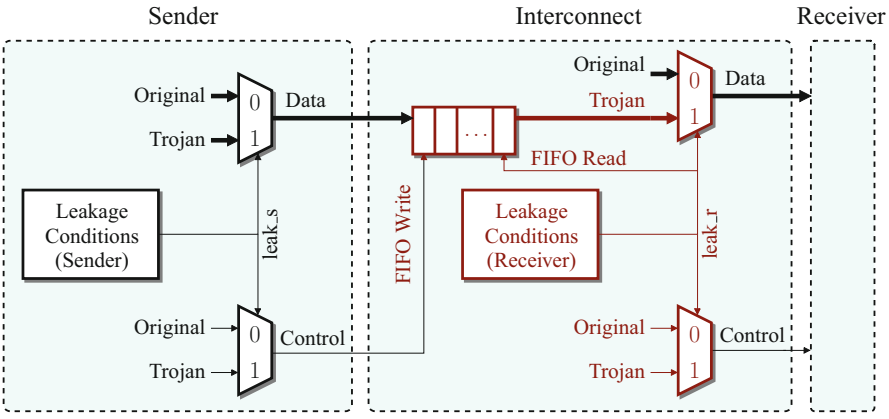

Fig. 12.8 Trojan channel circuitry

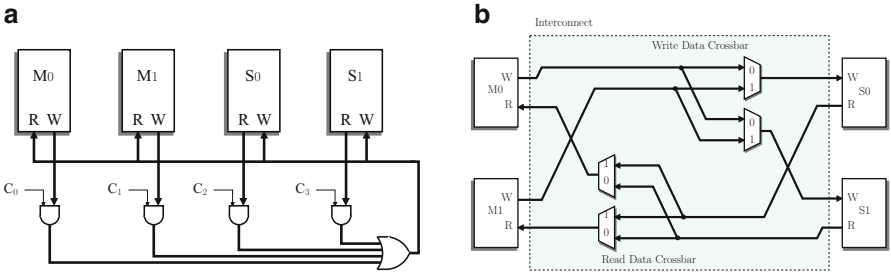

Fig. 12.9 Bus topologies on opposite ends of the area v. throughput spectrum. (**a**) Shared R/W data channels [44]. (**b**) Concurrent data channels [47]

interfaces to pass data from the sender to the receiver. For example, the line labeled Data in Fig. 12.8 on the sender's side could be the write data or read/write address port if the sender is a bus master and the read data port if the sender is a bus slave and vice versa for the Data on the receiver's side.

Since the Trojan data is transmitted using the same lines as normal bus traffic, additional signaling must mark when valid Trojan data is being transmitted. These signals are labeled as Control in Fig. 12.8, and like the Trojan data, are mapped to pre-existing data/address/control signals, meaning no additional interface ports are created. The Leakage Conditions Logic is protocol dependent and examines signals at the sender's interconnect interface to determine when it is "safe" to replace the original bus signal values with Trojan values.

12.4.2.1 Topology Dependent Trojan Channel Properties

All bus signals can be classified as address, data, or control signals, and additionally classified as belonging to read and/or write functionality. The interconnect topology specifies the degree of parallelism between the different categories of bus signals, and the connectivity between masters and slaves [44].

Figure 12.9a and b show the read and write data channels for topologies sitting at opposite ends of the area efficiency and channel throughput trade-off. Figure 12.9a is the most area efficient, but can only support a single transaction at a time, whereas Fig. 12.9b contains significantly more circuitry, but can support multiple simultaneous transactions.

In Fig. 12.9a, all read and write transactions are visible to all bus components, meaning no Trojan circuitry is required to simply snoop bus data. If a Trojan bus component wishes to send information, the black-colored circuitry inside the sender block of Fig. 12.8 is required. In Fig. 12.9b, data is not visible to a component uninvolved in the transaction. Unlike Fig. 12.9a, forming a channel between two slaves or two masters requires extra circuitry inside the interconnect, shown in red in Fig. 12.8. Because the signals at the sender's interconnect interface are not visible at the receiver's interface and vice versa, new leakage conditions are required, which monitor the receiver's interface and determine when it is safe to leak data without altering valid bus transactions. Signals available at the receiver's interface must also be selected to implement the Data and Control lines. The FIFO is necessary because leakage conditions at the sender and receiver may not occur simultaneously.

12.4.2.2 Protocol Dependent Trojan Channel Properties

The specifics of the Leakage Conditions Logic, which produces *leak_s* and *leak_r*, and the selection of Data and Control signals depend on the bus protocol used. Because of the similarities between various bus protocols, a general procedure for determining the Leakage Conditions Logic and the selection of Data and Control signals can be given.

Data Signal Selection In order to remain stealthy, the Trojan cannot create additional signals to transmit data, and must send data via pre-existing signals in the bus protocol. Being that the primary purpose of a bus is to transmit data, all bus protocol/topology combinations have signals that are suitable for sending/receiving Trojan data. In a protocol with separate read and write data signals, selection depends on if the Trojan Sender/Receiver resides in a master or slave component, since masters drive write data and observe read data signals, and vice versa for slave components. If the Trojan Sender resides in a master component, the read and write address signals can also be used to send Trojan data.

Leakage Conditions Logic Since pre-existing bus signals are used to transmit Trojan data, logic ensuring that normal bus operation is not compromised by the Trojan is necessary. The Leakage Conditions Logic examines protocol control signals to identify when Trojan Data signals are not being used to transmit *valid* data, and have unspecified values. Every bus protocol clearly defines the conditions for which data, address, and error reporting signals are valid. Some protocols, such as AXI4, designate a "valid" signal for each data channel, while others such as APB use the current state within the protocol to identify which signals are valid. *leak_s* is set when the Trojan Sender has data to transmit and the Data signals are not involved in a valid transaction. If the Trojan Sender is leaking valid bus transactions instead of actively sending information, then *leak_s* is not needed. *leak_r* is set when there are items in the Trojan FIFO and the Data signals at the receiver interface are not currently involved in a valid transaction.

Control Signal Selection When a Trojan Data signal is not being used in a valid bus transaction, its value is unspecified. During idle bus cycles, either Trojan data is being transmitted, or the bus is truly idle, and no data (Trojan or valid) is sent. To distinguish between these two cases, existing bus signals are selected to be Trojan Control signals, which mark when Trojan data is on the bus. The criteria for selecting these signals and their corresponding values is that when *leak_s/leak_r* is asserted, the normal behavior of the signal is predictable, but also unspecified. For most protocols, control signals are good candidates because they often are unused during idle cycles, yet their values remain static when idle for a given implementation.

We refer the reader to [22] for a detailed comprehensive presentation of the selection of Trojan Data and Control signals and Leakage Conditions Logic for the AMBA AXI4 protocol, and instead explain the reasoning behind a single selection of these signals for the AXI4-Lite based example system in the following section.

12.4.3 AXI4-Lite Interconnect Trojan Example

The system shown in Fig. 12.10 is created to verify the AXI4-Lite Interconnect Fabric through RTL simulation. The two slaves are simple 8-bit adder coprocessors which receive three operands to add via the interconnect from three processors.

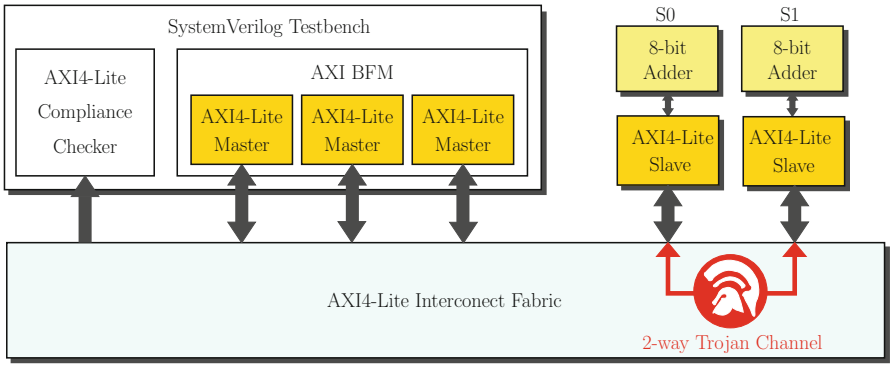

Fig. 12.10 AXI4-lite example system verification infrastructure

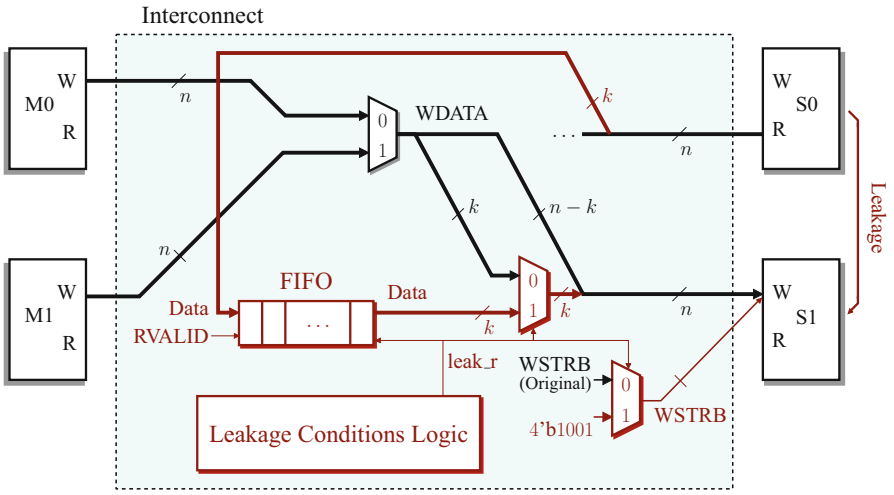

Fig. 12.11 Trojan channel logic for AXI4-lite interconnect

Since the specifics of the main processors are irrelevant, in the example infrastructure, they are replaced by AXI4-Lite bus functional models (BFMs) from [48]. Additionally, AXI4-Lite assertions packaged by ARM for protocol compliance checking [49] are active during system simulation.

The AXI4-Lite Interconnect Fabric IP block used is the LogiCORE IP AXI Interconnect (v1.02.a) from Xilinx [47] configured in Shared-Address Multiple-Data (SAMD) mode (the topology shown in Fig. 12.9b).

The AXI4-Lite Interconnect IP in Fig. 12.10 is infected with two copies of the circuitry shown in red in Fig. 12.11 to allow S1 to snoop on read requests for S0 and vice versa. Without the Trojan, the read data channel for S0 is not visible to S1 and vice versa. We now provide the reasoning behind the selection of Data and Control signals and Leakage Conditions Logic for the Trojan channel circuitry shown in Fig. 12.11.

Data Signal Selection k-bits of the n-bit AXI4-Lite write data signal (WDATA) are chosen to transfer the leaked information stored in the FIFO to the bus interface at S1 because S1 is a slave component, driving data signals on the read data channel, and receiving signals on the write data channel. There is no Trojan Data signal for the Sender since k bits of valid read data from S0 are being snooped, not actively transmitted.

Leakage Conditions Logic In AXI4 and AXI4-Lite 5 independent transaction channels are seen at the interface of every master and slave: the read address channel, read data channel, write address channel, write data channel, and write response channel [45]. Each channel uses a VALID/READY handshake signal pair, as shown in Fig. 12.7, to indicate when the receiver is ready to process bus data, and to mark when valid data is on the bus. Since the Trojan Data is being transmitted using the write data signal, the write data channel valid signal (WVALID) can be used to define *leak_r* as follows: $leak\_r = troj\_data\_ready$ & !WVALID.

Control Signal Selection The WSTRB signal is used to indicate when leaked data is on the bus. Normally WSTRB $==$ 1, because the bus data width is 32-bits, however the adder coprocessors have registers which are 8-bits wide. Because of this, any values for WSTRB with Hamming weight greater than 1 have no functional relevance in this system and are unused, making such values good candidates for marking when leaked data is on the bus. In this example Trojan channel, when information is leaked on WDATA, WSTRB $==$ 9.

The waveform in Fig. 12.12 first demonstrates how 3 read data responses (values 42, 15, then 14) from S1 are snooped and routed to S0's write channel, then shows a single read data response (value 96) from S0 routed to S1's write channel, and finally another read data response from S1 (value 13) leaked to S0. All Trojan transactions are highlighted in red in Fig. 12.12.

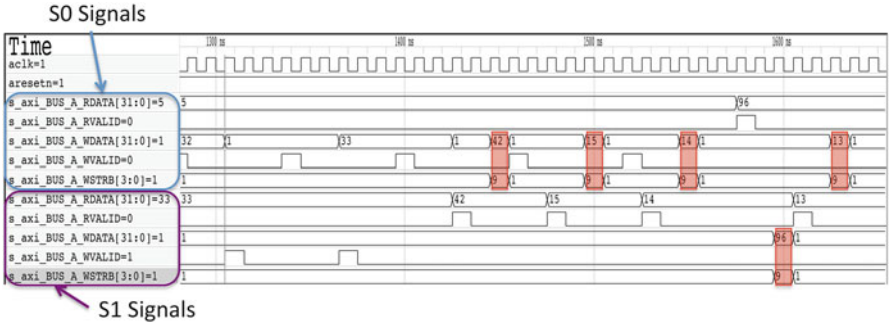

Fig. 12.12 2-Way information leakage waveform

Table 12.4 Trojan-free design results (after place and route)

Configuration	# FF	# LUT	# BRAM	Frequency [MHz]
3 masters 2 slaves	1814	2474	2	250
4 masters 6 slaves	3071	4247	3	250

Table 12.5 Area overhead of 2-way HW-Trojan channel

Data width	FIFO depth	% increase in FF		% increase in LUT	
		3M2S	4M6S	3M2S	4M6S
2	2	0.8	0.5	0.9	0.4
	4	1.1	0.7	1.5	0.6
	8	1.4	0.8	1.8	1.1
4	2	1.0	0.6	1.4	0.7
	4	1.3	0.8	2.0	0.8
	8	1.7	1.0	2.0	1.5
8	2	1.4	0.8	1.8	1.0
	4	1.8	1.0	2.4	1.2
	8	2.1	1.2	3.0	1.7

For AXI4-Lite, there are over 50 assertions monitoring bus signals during simulation, and none of them are violated even when information is flowing through the Trojan channel.

12.4.3.1 Overhead

To determine the area and timing overhead of implementing a 2-way Trojan channel between S0 and S1, the SystemVerilog Testbench in Fig. 12.10 is replaced by several simple bus masters. Table 12.4 shows results for the Trojan-free design, after placement and route, assuming 3 masters and 2 slaves (labeled as 3M2S) as well as 4 masters and 6 slaves (labeled as 4M6S) for a Virtex-7 FPGA (7vx330t-3).

Table 12.5 illustrates how the selection of Trojan channel parameters Data Width (k) and FIFO Depth (d) affects the results. The Trojan channel does not affect the operating frequency of the design, and stays within 3 % of the original FF and LUT utilization. As the number of masters and slaves increases, the interconnect and overall design area increases, but the size of the Trojan circuitry does not change.

The Trojan channel is easier to hide as the complexity of the interconnect and the number of components connected increases. The master and slave components used to generate the results in Tables 12.4 and 12.5 are far simpler than those in a typical SoC, so the results in Table 12.5 give a loose upper bound on the expected percentage of area increase caused by the Trojan channel in a modern design.

12.5 Conclusion

In this chapter we have addressed the threat of Hardware Trojans in unspecified design functionality. Due to the complexity of modern chips, a design specification usually only defines a small fraction of behavior. Traditional verification techniques only focus on ensuring the correctness of specified behavior, meaning any modifications or bugs (malicious or accidental) only affecting unspecified functionality will likely go undetected.

We have shown how this verification hole allows an attacker with the ability to modify the design to stealthily undermine the security of a system. We have shown that all secret key bits in an Elliptic Curve Processor can be leaked by only modifying RTL don't cares, and that it is possible to create a covert Trojan communication channel on top of existing on-chip bus infrastructure for common bus protocols by only modifying the on-chip bus interface signals when the channel is idle.

By viewing security as an extension of the verification problem we develop several analysis methodologies based on existing techniques such as equivalence checking and mutation testing which both prevent Trojans and increase confidence in the correctness of specified design functionality. These techniques include a Trojan prevention methodology based on equivalence checking that classifies all don't care bits in a design as dangerous or safe and a mutation testing based methodology capable of identifying dangerous unspecified functionality regardless of the abstraction level or class of design analyzed. Because unspecified functionality is by nature unknown, there is still much work to be done in fully exploring the scope of the Trojan threat in this space.

References

1. M. Dale, Verification crisis: managing complexity in SoC designs. EE Times (2001) [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1215507
2. S. Adee, The hunt for the kill switch. IEEE Spectr. **45**(5), 34–39 (2008)
3. S. Mitra, H.-S.P. Wong, S. Wong, The Trojan-proof chip. IEEE Spectr. **52**(2), 46–51 (2015)
4. Y. Shiyanovskii, F. Wolff, A. Rajendran, C. Papachristou, D. Weyer, W. Clay, Process reliability based trojans through NBTI and HCI effects, in *2010 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)* (IEEE, Anaheim, 2010), pp. 215–222
5. L.-W. Kim, J.D. Villasenor, Ç.K. Koç, A Trojan-resistant system-on-chip bus architecture, in *Proceedings of the 28th IEEE Conference on Military Communications*. Ser. MILCOM'09 (2009), pp. 2452–2457
6. S.T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, Y. Zhou, Designing and implementing malicious hardware, in *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (USENIX Association, Berkeley, CA, 2008), pp. 5:1–5:8
7. L. Lin, W. Burleson, C. Paar, Moles: malicious off-chip leakage enabled by side-channels, in *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*, November (2009), pp. 117–122

8. G.T. Becker, F. Regazzoni, C. Paar, W.P. Burleson, Stealthy dopant-level hardware trojans, in *Cryptographic Hardware and Embedded Systems (CHES)*. Ser. Lecture Notes in Computer Science, vol. 8086, ed. by G. Bertoni, J.-S. Coron (Springer, Berlin, Heidelberg, 2013), pp. 197–214
9. M. Tehranipoor, F. Koushanfar, A survey of hardware trojan taxonomy and detection. IEEE Des. Test Comput. **27**(1), 10–25 (2010)
10. R.S. Chakraborty, S. Narasimhan, S. Bhunia, Hardware trojan: threats and emerging solutions, in *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International* (IEEE, San Francisco, 2009), pp. 166–171
11. C. Krieg, A. Dabrowski, H. Hobel, K. Krombholz, E. Weippl, Hardware malware. Synth. Lect. Inf. Secur. Priv. Trust **4**(2), 1–115 (2013)
12. A. Waksman, S. Sethumadhavan, Silencing hardware backdoors, in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, Ser. SP'11 (2011), pp. 49–63
13. S.S. Ali, R.S. Chakraborty, D. Mukhopadhyay, S. Bhunia, Multi-level attacks: an emerging security concern for cryptographic hardware, in *2011 Design, Automation Test in Europe* (2011), pp. 1–4
14. S. Bhasin, J.L. Danger, S. Guilley, X.T. Ngo, L. Sauvage, Hardware Trojan horses in cryptographic IP cores, in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, August (2013), pp. 15–29
15. D. Agrawal, et al., Trojan detection using IC fingerprinting, in *IEEE Symposium on Security and Privacy*, 2007
16. A. Waksman, M. Suozzo, S. Sethumadhavan, FANCI: Identification of stealthy malicious logic using Boolean functional analysis, in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS'13* (ACM, New York, 2013), pp. 697–708
17. D. Sullivan, J. Biggers, G. Zhu, S. Zhang, Y. Jin, FIGHT-metric: functional identification of gate-level hardware trustworthiness, in *Proceedings of the 51st Annual Design Automation Conference, DAC'14* (ACM, New York, 2014), pp. 173:1–173:4
18. J. Zhang, F. Yuan, L. Wei, Z. Sun, Q. Xu, VeriTrust: verification for hardware trust, in *Proceedings of the 50th Annual Design Automation Conference, DAC'13* (ACM, 2013), pp. 61:1–61:8
19. M. Hicks, et al., Overcoming an untrusted computing base: detecting and removing malicious hardware automatically, in *Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP'10* (IEEE Computer Society, Washington, 2010), pp. 159–172
20. N. Fern, S. Kulkarni, K.-T. Cheng, Hardware Trojans hidden in RTL don't cares - Automated insertion and prevention methodologies, in *Test Conference (ITC), IEEE International*, October (2015), pp. 1–8
21. N. Fern, K.-T. Cheng, Detecting hardware trojans in unspecified functionality using mutation testing, in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD'15* (IEEE Press, Piscataway, 2015), pp. 560–566
22. N. Fern, I. San, Ç.K. Koç, K.T. Cheng, Hardware Trojans in incompletely specified on-chip bus systems, in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March (2016), pp. 527–530
23. R.A. Bergamaschi, D. Brand, L. Stok, M. Berkelaar, S. Prakash, Efficient use of large don't cares in high-level and logic synthesis, in *1995 IEEE/ACM International Conference on Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers* , November (1995), pp. 272–278
24. M. Turpin, The dangers of living with an x (bugs hidden in your verilog), in *Boston Synopsys Users Group (SNUG)*, October 2003
25. L. Piper, V. Vimjam, X-propagation woes: masking bugs at RTL and unnecessary debug at the netlist, in *Design and Verification Conference and Exhibition (DVCon)*, 2012
26. H.Z. Chou, H. Yu, K.H. Chang, D. Dobbyn, S.Y. Kuo, Finding reset nondeterminism in rtl designs - scalable x-analysis methodology and case study, in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March (2010), pp. 1494–1499

27. Cadence conformal equivalence checker [Online]. Available: http://www.cadence.com/products/ld/equivalence_checker
28. M. Turpin, Solving verilog x-issues by sequentially comparing a design with itself. you'll never trust unix diff again! in *Boston Synopsys Users Group (SNUG)*, 2005
29. A.R. Bradley, SAT-based model checking without unrolling, in *Verification, Model Checking, and Abstract Interpretation* (Springer, Berlin/Heidelberg 2011), pp. 70–87
30. G. Cabodi, S. Nocco, S. Quer, Improving SAT-based bounded model checking by means of BDD-based approximate traversals, in *Design, Automation and Test in Europe Conference and Exhibition, 2003* (2003), pp. 898–903
31. J.W. Bos, J.A. Halderman, N. Heninger, J. Moore, M. Naehrig, E. Wustrow, Elliptic curve cryptography in practice, in *Financial Cryptography and Data Security* (Springer, 2014), pp. 157–175
32. C. Rebeiro and D. Mukhopadhyay, High performance elliptic curve crypto-processor for FPGA platforms, in *12th IEEE VLSI Design and Test Symposium*, 2008
33. ABC [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/
34. Atrenta spyglass lint tool [Online]. Available: http://www.atrenta.com/pg/2/
35. Y. Jia and M. Harman, An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
36. B. Breech, M. Tegtmeyer, L. Pollock, An attack simulator for systematically testing program-based security mechanisms, in *2006 17th International Symposium on Software Reliability Engineering*, November (2006), pp. 136–145
37. N. Bombieri, F. Fummi, G. Pravadelli, M. Hampton, F. Letombe, Functional qualification of tlm verification, in *2009 Design, Automation Test in Europe Conference Exhibition*, April (2009), pp. 190–195
38. P. Lisherness, K.T. Cheng, Scemit: a systemc error and mutation injection tool, in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, June (2010), pp. 228–233
39. N. Bombieri, F. Fummi, G. Pravadelli, A mutation model for the systemC TLM 2.0 communication interfaces, in *2008 Design, Automation and Test in Europe*, March (2008), pp. 396–401
40. Synopsys certitude [Online]. Available: https://www.synopsys.com/TOOLS/VERIFICATION/FUNCTIONALVERIFICATION/Pages/certitude-ds.aspx
41. P. Lisherness, N. Lesperance, K.T. Cheng, Mutation analysis with coverage discounting, in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March (2013), pp. 31–34
42. UART 16550 core [Online]. Available: http://opencores.org/project,uart16550
43. Wishbone bus [Online]. Available: http://opencores.org/opencores,wishbone
44. S. Pasricha, N. Dutt, *On-Chip Communication Architectures: System on Chip Interconnect* (Morgan Kaufmann Publishers Inc., Burlington, 2008)
45. *AMBA AXI and ACE Protocol Specification, Issue E*, ARM, 2013
46. L.-W. Kim, J.D. Villasenor, A system-on-chip bus architecture for thwarting integrated circuit Trojan horses, in *IEEE Transactions on VLSI Systems* **19**(10), 1921–1926 (2011)
47. *DS768: LogiCORE IP AXI Interconnect (v1.02.a)*, Xilinx Inc., March 2011
48. Axi4 bfm [Online]. Available: https://github.com/sjaeckel/axi-bfm
49. Amba 4 axi4, axi4-lite and axi4-stream protocol assertions bp063 release note (r0p1-00rel0), ARM [Online]. Available: https://silver.arm.com/browse/BP063