Detecting Hardware Trojans in Unspecified Functionality Through Solving Satisfiability Problems

Nicole Fern
UC Santa Barbara, USA
nicole@ece.ucsb.edu

Ismail San
Anadolu University, Turkey
isan@anadolu.edu.tr

Kwang-Ting (Tim) Cheng
HKUST, Hong Kong
timcheng@ust.hk

Abstract— For modern complex designs it is impossible to fully specify design behavior, and only feasible to verify functionally meaningful scenarios. Hardware Trojans modifying *only* unspecified functionality are not possible to detect using existing verification methodologies and Trojan detection strategies. We propose a detection methodology for these Trojans by 1) precisely defining "suspicious" unspecified functionality in terms of information leakage, and 2) formulating detection as a satisfiability problem that can take advantage of the recent advances in both boolean and satisfiability modulo theory (SMT) solvers. The formulated detection procedure can be applied to a gate-level design using commercial equivalence checking tools, or directly to the Verilog/VHDL code by reasoning about the satisfiability of SMT expressions built from traversing the data-flow graph. We demonstrate the effectiveness of our approach on an adder coprocessor and a UART communication controller infected with Trojans which process information leaked from the on-chip bus during idle cycles using signals with only partially specified behavior.

I. Introduction

Hardware Trojans are malicious modifications made to a circuit during any stage in the design life cycle. Recently Trojans have become a concern for both semiconductor design houses and the U.S. government due to the fact that the design, manufacturing, testing, and deployment of silicon chips involves many companies and countries [1]. Hardware Trojans may be inserted into the system specification, high-level models, RTL code, gate level net list, circuit layout, or circuit mask for a given design. Trojan behavior ranges 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 or leak information through side channels [2].

Trojans are classified according to their triggering mechanism (conditions which cause the malicious circuitry to become active) and payload (functionality). Trojan payloads either affect logic functions of some design signals or leak information through side-channels such as power and timing. The Trojans which affect signal logic typically violate design specifications, therefore rely on rare triggering conditions to suppresses Trojan behavior until the condition is satisfied to avoid detection.

The Trojans addressed in this work do not rely on rare triggering conditions to stay hidden, but instead **only** alter the logic functions of design signals which have **unspecified behavior**. This means the Trojan **never** violates the design specification and will not be detected by both traditional verification techniques focused on conformance checking or Trojan detection methodologies targeting the identification of rarely activated circuitry.

Our work formulates the detection problem in terms of information leakage by observing that if a signal, x, is unspecified under a condition, $\mathcal{C}$, the value of x should not propagate to important points in the design such as registers or primary outputs, and if it does, Trojan circuitry is present or a design bug exists. This observation can be concisely expressed as a satisfiability problem to take advantage of the recent advances in both boolean and satisfiability modulo theory (SMT) solvers.

Our main contributions are 1) a general and complete detection methodology for Trojans in unspecified functionality which can be followed using a wide variety of tools and techniques, 2) a precise formulation for "dangerous" unspecified functionality expressed as a satisfiability problem, and 3) a method to detect the Trojan communication channels proposed in [3] which are formed by only modifying signals in common on-chip bus protocols when they are unspecified.

The rest of the paper is organized as follows: Section II explores related work, Section III gives the threat model, Section IV precisely formulates the detection problem, Section V provides our detection methodology in detail, Section VI demonstrates the effectiveness and quantifies the overhead of our method using several example designs, and we conclude in Section VII.

II. Related Work

Trojans with Rare Triggering Conditions: Many Trojans hide from the verification effort by only performing malicious functionality under extremely rare triggering conditions (ex. [4, 5]). Trojans with these triggering mechanisms generally deploy a payload which clearly violates system specifications. Existing pre-silicon Trojan detection methods (both static and simulation-based) focus on identifying logic activated by this triggering circuitry in the RTL code or gate-level netlist [6, 7, 8, 9]. These methods identify "almost unused" logic, where rareness is quantified by an occurrence probability threshold. The Trojans we address in this work never violate design specifications, so have no need to hide in almost unused logic, meaning existing methods are unable to highlight circuitry affected by this Trojan type.

Trojans in Unspecified Functionality: Several works propose and address Trojans in unspecified functionality [3, 10, 11], however none provide detection methodologies, only prevention techniques which must be applied to a Trojan-free design [10] and identification of unspecified design functionality

which *could potentially* be modified by an attacker [11]. Our method is the first to provide the ability to actually classify unspecified functionality as being Trojan-infected or Trojan-free.

In [3], Trojans are proposed which only modify signals in common on-chip bus protocols when they are unspecified (usually during idle cycles) giving the attacker a powerful mechanism to move information covertly through an entire SoC, but no methods are provided to detect these Trojans. A major contribution of our work is providing a detection mechanism for these Trojans and demonstrating its effectiveness on several designs infected with Trojan communication channels.

Information Flow Analysis: Information flow analysis techniques verify security properties such as confidentiality, integrity, and availability. There exist several methods for analysis of information flow in hardware [12, 13, 14] and firmware [15] as well as Trojan detection methods which tag assets such as cryptographic key bits in the design then use formal methods such as model checking [16] and theorem proving [17] to analyze how these bits propagate through the design.

Information flow properties specify the conditions under which information can safely flow between signals in a design. A whitelist of such properties describes the proper access and disclosure mechanisms for all important signals, and the design is analyzed to detect violations of the specified properties.

Our analysis instead focuses on creating a list of signals and the conditions under which they are functionally meaningless, then making the observation that it is suspicious for information to flow from a signal *anywhere* in the design when it is unspecified. The threat model addressed is different, but our problem can be formulated in terms of information leakage properties. In this work we focus on providing a concise theoretical formulation for Trojan detection, and provide several ways (but certainly not all possible ways) to solve the problem.

III. Threat Model

Our method detects Trojans which exploit design inputs or internal signals when they are unspecified to modify other design signals in a malicious, but covert manner.

For example, consider a peripheral with registers visible to unprivileged software connected to the same on-chip bus as a memory controller. The Trojans proposed in [3] allow an attacker to leak information such as memory accesses from the root user to the bus interface of this easily accessible peripheral when signals in the interface are not being used for valid bus transactions. The peripheral could then transfer the leaked information to unused addresses in the register space or unused bit fields in existing registers allowing a malicious, but unprivileged, software program to access sensitive data.

It should be noted that this work does not address Trojans which leak information by modifying circuit side-channels such as power consumption. We only address Trojans which change logic values of design signals.

Trojan Insertion Stage: It is assumed that *no golden RTL model exists* to aid in Trojan detection during later stages in the design cycle, meaning that it is possible for Trojans to be inserted in the RTL code or higher-level model. Our detection methodology exists to increase confidence that the RTL model is Trojan-free and bug-free. Nowadays, hundreds of engineers

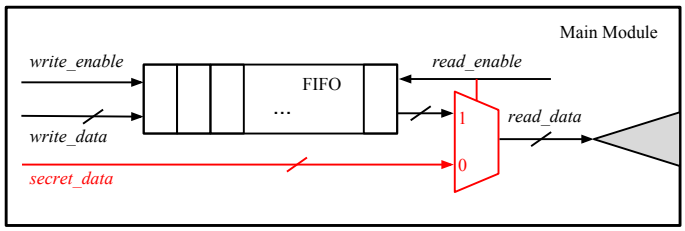

Fig. 1. Trojan-infected FIFO (Trojan Circuitry affecting unspecified functionality shown in red)

are involved in the design and test of a silicon chip. A single malicious 3rd party IP provider, CAD tool vendor, or disgruntled engineer has the ability to insert a Trojan in the RTL code.

IV. Problem Formulation

For a hardware design f, let x be a signal in f which is **unspecified** under condition $\mathcal{C}$. For a given hardware design, there will be many (x, $\mathcal{C}$) pairs. Trojans will insert malicious functionality by modifying x during $\mathcal{C}$.

Key insight: Instead of enumerating and targeting every possible Trojan in this space, we observe that any functionality the attacker inserts must eventually influence design outputs or modify design state otherwise it is redundant. While malicious functionality is not redundant, by definition unspecified functionality *should be*, and this allows us to formulate the problem without modeling Trojan functionality or defining "expected" behavior for the unspecified functionality. The only assumption made is that when x is unspecified, its value should never influence any key points in the design.

We determine if x can influence circuit output under $\mathcal{C}$ by precisely formulating this question as a satisfiability problem. If Equation 1 is **satisfiable**, two different values of x (x_0 and x_1) result in differences in circuit output under $\mathcal{C}$, which is not consistent with properties of unspecified functionality meaning the existence of Trojan circuitry or a design bug is likely.

$$\mathcal{C} \wedge (f_{x \to x_0} \oplus f_{x \to x_1}) \tag{1}$$

A Motivating Example: Figure 1 shows a simple FIFO embedded in a larger design. In both the Trojan-free and Trojan-infected versions, when $read\_enable = 1$, the $read\_data$ signal is correctly assigned output from the FIFO, but when $read\_enable = 0$, the Trojan circuitry assigns the value of $secret\_data$ to $read\_data$, allowing an attacker who can observe the main module's outputs to learn the value of $secret\_data$ since the $read\_data$ signal propagates to the boundaries of the design.

Because the Trojan circuitry never affects FIFO read functionality, it is unlikely to be detected by existing verification methodologies. The value of $read\_data$ when $read\_enable = 0$ is unspecified because it is assumed that any circuitry in the fan-out of $read\_data$ will only propagate or store its value when a valid FIFO read occurs. It would be a waste of precious verification resources to specify and verify the value of $read\_data$ when $read\_enable = 0$, however verifying that the FIFO is Trojan-free only involves defining the (x,

$\mathcal{C}$) pairs ($read\_data$, $read\_enable = 0$) and ($write\_data$, $write\_enable = 0$) and determining the satisfiability of Equation 1 for each pair.

Identifying (x, $\mathcal{C}$) Pairs: Identifying (x, $\mathcal{C}$) pairs for a design is not yet a solved problem. An automated approach, which we plan to elaborate on in future work, uses the mutation testing based technique proposed in [11] in which **undetected** mutants affecting attacker observable signals are flagged as dangerous because an attacker can modify behavior related to the mutant to leak information. A signal differing under a dangerous mutant, and the corresponding condition under which the difference occurs form an (x, $\mathcal{C}$) pair.

Because mutation testing is expensive, another approach is to define (x, $\mathcal{C}$) pairs manually for specific classes of designs. Identifying a complete list of (x, $\mathcal{C}$) pairs for a complex design may not always be possible, but more pairs included for analysis results in greater confidence in the security of the design. For designs with on-chip bus interfaces, it is straightforward to comprehensively list (x, $\mathcal{C}$) pairs involving bus signals for several well known standard bus protocols. The example designs used to demonstrate the effectiveness of our detection method in Section VI give details for the AXI4-Lite and Wishbone protocols.

V. Detection Methodology

Before the analysis of any (x, $\mathcal{C}$) pairs, an SMT or boolean formula, o, is built for each primary output by traversing the data-flow graph or synthesizing the design.
Then for each (x, $\mathcal{C}$) pair, and primary output o in f:

1. Replace x with new variables x_0 and x_1 respectively in 2 identical copies of o
2. If $SAT(\mathcal{C} \wedge (o_{x \to x_0} \oplus o_{x \to x_1}))$, flag signal x as *dangerous* for involvement in Trojan circuitry
3. Inspect design behavior and code activated by the satisfying assignment

If f is an RTL design written in Verilog or VHDL, the analysis can be performed on 1) formulas containing only boolean variables and logical connectives, or 2) formulas containing symbols and operators whose semantic meaning is derived from a theory (ex. the $+$, $-$, $\times$, $<$, $\geq$ operators all have behavior described by the theory of arithmetic and commonly appear in Verilog/VHDL code).

Determining the satisfiability of formulas containing symbols and operators governed by the theory of integers, bit vectors, and arrays requires Satisfiability Modulo Theory (SMT) solvers [18]. In the following section we detail how to construct SMT formulas for each circuit output directly from Verilog code, *meaning no synthesis tool is required*, and use PySMT [19] to perform Steps 1 and 2 in the detection procedure.

The satisfiability of boolean formulas can be determined using a SAT solver, but the approach we present in Section V-B leverages the robustness and scalability of commercial equivalence checking tools (ex. Cadence Conformal [20]) in which a gate-level model for f produced using logic synthesis is analyzed.

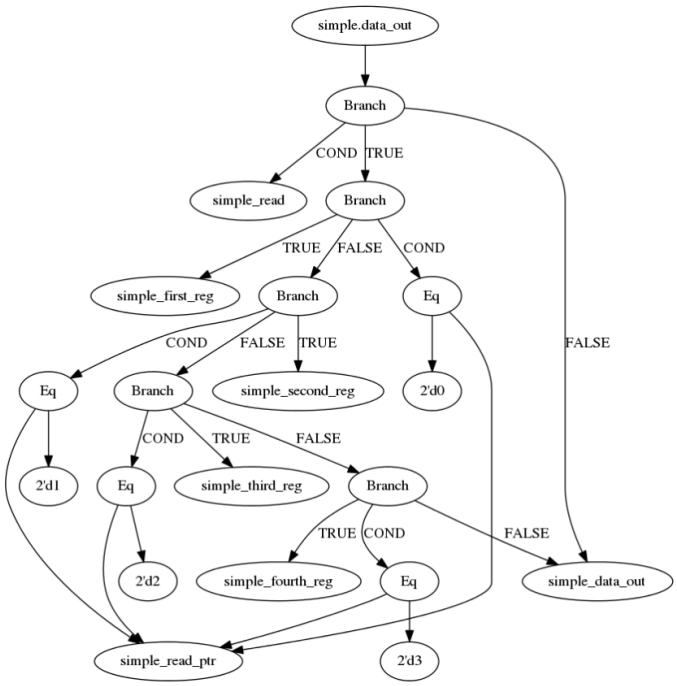

Fig. 2. Data-flow Graph for $simple.data\_out$ Generated by Pyverilog

A. SMT Formulas from RTL Code

Pyverilog [21], an open-source Verilog code parser and static analysis tool written in Python is used to build a data-flow graph for each primary output. To construct the data-flow graph, Pyverilog first builds an abstract syntax tree (AST) representation from the Verilog code then traverses the AST twice to determine signal scope, resolve all parameters and constants in the design hierarchy, and create an assignment tree for every signal describing the complete data-flow.

Figure 2 shows the data-flow graph generated by Pyverilog for the signal $data\_out$. The Verilog code for $data\_out$ is given in Listing 1. The first branch node corresponds to the *if* statement on Line 14 of Listing 1. The remaining branch nodes correspond to the *case* statment.

Listing 1 simple.v

```
1  module simple(clk, rst, data_in, read,
       write, data_out, out_valid);
2
3    input clk, rst, read, write;
4    input [7:0] data_in;
5    output reg [7:0] data_out;
6    output reg out_valid;
7
8    wire [1:0] read_ptr, write_ptr;
9    reg [7:0] first_reg, second_reg,
       third_reg, fourth_reg;
10   ...
11
12   //data_out logic
13   always @(posedge clk) begin
14     if (read)
15       case(read_ptr)
```

```
16          2'b00: data_out <= first_reg;
17          2'b01: data_out <= second_reg;
18          2'b10: data_out <= third_reg;
19          2'b11: data_out <= fourth_reg;
20        endcase
21    end
22    ...
23  endmodule
```

Data-flow graph nodes fall into the following categories: branches, operators (logic, arithmetic, etc.), bit vector slicing and concatenation, constants, and terminals. Our detection methodology builds the formula for each output, o, using Python functions provided by PySMT [19] to create symbols/-variables, and describe bit vector, arithmetic, and boolean operations based on the nodes encountered in the data-flow graph.

PySMT provides functions to build formulas involving several theories such as Linear Real Arithmetic (LRA), Real Difference Logic (RDL), Equalities and Uninterpreted Functions (EUF), and Bit-Vectors (BV), then calls existing solvers such as MathSAT, Z3, CVC4, Yices 2, CUDD, PicoSAT, and Boolector to determine if the formula is satisfiable based on the theories present [19].

For each output, all data-flow graph nodes generated by Pyverilog are processed by a recursive traversal procedure which returns sub-formulas for each node. Branches, operators, constants, and bit vector operation nodes are straightforward, as PySMT has good support for constructing expressions with boolean and arithmetic bit-vector operations. New variables are created when terminal nodes are encountered.

Terminal nodes in the graph correspond to intermediate *reg* and *wire* Verilog variables in all modules, inputs and outputs for modules instantiated by the top module, and primary inputs for the top module. If the terminal is a primary input for the top module, a new formula variable is either created or retrieved from a table containing formula variables already encountered during processing. Otherwise, a formula for the *reg*/*wire* variable, input, or output is built by exploring its own data-flow graph and then stored in a table to avoid repeated analysis.

The formula built from traversing the graph in Figure 2 would contain the $simple.read$ variable because it is a primary input, but the nodes corresponding to $simple.read\_ptr$, $simple.first\_reg$, $simple.second\_reg$, etc. have their own data-flow graphs which are traversed to produce formulas in terms of primary inputs.

One should note that signals corresponding to state elements in the design will reference themselves in the data-flow graph. If this situation is encountered, the variable will become a symbol in the formula to avoid an infinite processing loop. Eventually, the formulas for primary outputs will only reference primary input and state variables.

PySMT functions are used to construct Equation 2, which determines how x affects output o under $\mathcal{C}$.

$$\mathcal{C} \wedge (o_{x \to x_0} \oplus o_{x \to x_1}) \tag{2}$$

The "xor" and "and" operators have corresponding functions in PySMT, and the `substitute` function is used to create $o_{x \to x_0}$ and $o_{x \to x_1}$, where x_0 and x_1 are new variables. If Equation 2 is satisfiable, PySMT provides a *model*, or one set of possible satisfying assignments for all variables in Equation 2 (including the new variables x_0 and x_1) proving that x can influence o under $\mathcal{C}$.

B. Equivalence Checking

The applicability of the Trojan detection procedure described in Section V-A to commercial hardware designs is limited by the robustness of the Pyverilog parser and the efficiency and usability of the available open-source SMT solvers. Logic equivalence checking for hardware designs is a mature, scalable, and robust technology. We present a way to perform the detection procedure using Cadence Conformal LEC [20], the advantage being employing a tool already part of the circuit design workflow.

A disadvantage of this approach is that if x is a multi-bit signal, the satisfiability of Equation 1 must be determined for each bit in x separately. Because equivalence checking compares two gate-level designs, it is impossible to symbolically replace x, a multi-bit signal, with x_0 in one version and x_1 in the other *unless* x is a single bit.

To determine the satisfiability of Equation 1 using Conformal, 1) two versions of the design must be created assigning a single bit in x to 0 in one version and 1 in the other, and 2) equivalence should be proven only under $\mathcal{C}$.

To form the two circuit versions, LEC has two commands: `add_primary_input` and `add_pin_constraints`, that can be used to force arbitrary signals in the design to 0 or 1 in the golden or revised versions of the circuit. Ignoring equivalence checking results under certain conditions is accomplished in LEC using the `$constraint` function. For example, if a and b are design signals, inserting `$constraint(a==1 && b==1)` in the design source code forces LEC to ignore counterexamples requiring a or b to be 0.

VI. Case Studies

To validate our detection methodology, we infect two designs, 1) an adder coprocessor, and 2) a Universal Asynchronous Receiver/Transmitter (UART) communication controller, with Trojans modifying unspecified functionality to leak information. The inserted Trojans covertly receive information from the bus interface during idle cycles, and then modify unspecified design functionality either to store the information for later retrieval by the attacker or transmit the information outside the chip. The Trojans created for the case studies are non-trivial and representative of malicious functionality that can be inserted into any bus peripheral. Critical information passes through the on-chip bus, making it a high value target for an attacker.

We apply both versions of the detection methodology (solving satisfiability of formula built from the data-flow graph and gate-level equivalence checking) to the designs in order to demonstrate the strengths and weaknesses of each approach. All Trojans are successfully identified by the method employing an SMT solver, and in Section VI-C we discuss the limitations of using equivalence checking tools for Trojan detection.

Table I summarizes the experimental results, giving the size of the two original and Trojan-infected designs in lines of code and 2-input NAND gates. The gate count was determined by

TABLE I
LINES OF CODE AND # 2-INPUT NAND GATES FOR EXAMPLE DESIGNS AND TOTAL ANALYSIS TIME FOR ALL $(x, \mathcal{C})$ PAIRS

Design	LOC		# 2NAND		Time (sec.)	
	Orig.	Trj.	Orig.	Trj.	Orig.	Trj.
Adder	614	616	839	877	0.61	0.69
UART	2269	2273	5829	5836	8.59	8.63

synthesizing the design using Synopsys Design Compiler (ver. I-2013.12-SP2) with a freely available 45nm technology library from NanGate [22]. Table I also lists the total time in seconds spent on Trojan detection. This includes parsing all design files and building the data-flow graph for each signal in addition to determining the satisfiability of Equation 2 for all $(x, \mathcal{C})$ pair and primary output combinations. The experiments were run on a Dell Optiplex 960 desktop computer with 8GB RAM.

A. Adder Coprocessor

The coprocessor takes input from an AMBA AXI4-Lite [23] bus interface to add 3 8-bit values: a, b, and c. The programmer communicates with the coprocessor by reading and writing to registers in the peripheral.

Trojan Description: Although the registers in the adder coprocessor are allocated 32-bits in the addressing scheme, many only use a single bit. For example, the user writes to the least significant bit of the *b_ap_vld* register to indicate that the b operand is valid, and the remaining bits go unused. The inserted Trojan circuitry takes advantage of this fact by storing 4 bits of leaked data from the AXI4-Lite bus interface in *b_ap_vld[5:1]* for a malicious software program to read. Unused register bits are an example of unspecified functionality common in many designs, making the inserted Trojan an ideal example to test our detection strategy.

It is assumed that the leaked data is otherwise not accessible by the adversary and they have access to registers in the coprocessor. The data is leaked using the Trojan communication channel proposed in [3], which only alters bus signals at the coprocessor's interface when they are not in use. While the Trojan modifies unspecified functionality present in the AXI4-Lite protocol, insertion of Trojan communication channels generalizes to all bus protocols and interconnect topologies.

The 4-bit Trojan data is delivered on the write data channel signal *WDATA[3:0]* when $\neg WVALID \wedge WSTRB == 4'b1111$. The Trojan residing in the coprocessor recognizes that when $WSTRB == 4'b1111$, data on the *WDATA* signal is from the Trojan communication channel then stores *WDATA[3:0]* in *b_ap_vld[5:1]*.

Description of $(x, \mathcal{C})$ Pairs: Input to the adder coprocessor module comes from the AXI4-Lite bus interface. AXI4 defines 5 independent transaction channels seen at the interface of every master and slave, with each channel employing a *VALID*/*READY* handshake signal pair to indicate when the receiver is ready to process bus data, and to mark when valid data is on the bus. We refer the reader to [23] for further detail on the AXI4-Lite protocol.

TABLE II
TROJAN DETECTION RESULTS FOR ADDER COPROCESSOR: ANALYSIS TIME IN SECONDS, AND OUTPUTS SAT FOR EACH $(x, \mathcal{C})$ PAIR

x	$\mathcal{C}$	Time, POs SAT	
		Orig.	Trj.
AWADDR	¬AWVALID	0.13	0.13
WDATA	¬WVALID	0.13	0.17, RDATA
WSTRB	¬WVALID	0.13	0.17, RDATA
ARADDR	¬ARVALID	0.10	0.10

From the perspective of Trojan detection, we wish to determine if any information received on the bus interface on any channel while the channel *VALID* signal is 0 can influence data read out from the registers of the adder coprocessor. Since the adder coprocessor is a bus slave, the inputs (x signals) are those shown in the first column of Table II, with $\mathcal{C}$ being the condition that the corresponding channel *VALID* signal is 0.

The signals listed in Table II cover all inputs to the adder coprocessor excluding clock and reset signals, meaning this list of $(x, \mathcal{C})$ pairs is complete for guaranteeing the adder is not processing leaked information.

Detection Results: The primary outputs for the adder coprocessor are the following: {*AWREADY, WREADY, BRESP, BVALID, ARREADY, RDATA, RRESP, RVALID, interrupt*}. The satisfiability of Equation 2 is determined for every output listed for every $(x, \mathcal{C})$ pair in Table II. Each row in Table II gives the time taken (in seconds) to build and analyze Equation 2 for all outputs using a specific $(x, \mathcal{C})$ pair and lists any outputs for which Equation 2 was satisfiable (if any) for both the Trojan-free and Trojan-infected versions of the adder.

For the Trojan-free design, **there were no false positives.** None of the $(x, \mathcal{C})$ pairs were able to influence primary outputs, which is consistent with our assumption that unspecified functionality should not be capable of significantly affecting design behavior. For the Trojan-infected design, our analysis was able to highlight the signals involved in the Trojan circuitry. *WDATA* and *WSTRB* are shown to influence *RDATA*, which is expected, because the Trojan alters the contents of the *b_ap_vld* register, which can be retrieved by a read request. The read and write address signals are not involved in the Trojan, and are not flagged by our detection methodology.

B. UART Communication Controller

We also validate our detection methodology on a UART circuit from OpenCores [24] designed to interface with a host processor through a Wishbone Bus Interface [25].

Trojan Description: To transmit data serially on the UART output stx_pad_o, 8 bits are written to the UART transmit data register through a Wishbone write transaction. Normally only a bus master is able to issue write requests and cause the UART to transmit data, however if a Trojan communication channel is inserted inside the Wishbone bus, it is possible for another slave to covertly signal the Trojan inserted in the UART controller, and cause data to be serially transmitted outside the chip.

The Trojan inserted in the UART allows writes to registers when $wb\_sel\_i == 4'b1001$. The Wishbone bus uses

TABLE III
TROJAN DETECTION RESULTS FOR UART CORE: ANALYSIS TIME IN SECONDS, AND OUTPUTS SAT FOR EACH (x, $\mathcal{C}$) PAIR

x	$\mathcal{C}$	Time, POs SAT	
		Orig.	Trj.
wb_adr_i	¬wb_stb_i ∨ ¬wb_cyc_i	1.54	1.63, (int_o, baud_o, dtr_pad_o, stx_pad_o, rts_pad_o)
wb_dat_i	¬wb_stb_i ∨ ¬wb_we_i ∨ ¬wb_cyc_i	3.00	2.82, (int_o, baud_o, dtr_pad_o, stx_pad_o, rts_pad_o)
wb_sel_i	¬wb_stb_i ∨ ¬wb_we_i ∨ ¬wb_cyc_i	2.95	3.07, (int_o, baud_o, dtr_pad_o, wb_ack_o, stx_pad_o, rts_pad_o)

32-bit data signals, but the UART registers are only 8-bits wide. wb_sel_i marks which byte lanes contain valid data. For the UART controller, any values of wb_sel_i with Hamming Weight > 1 are "don't care."

In addition to wb_sel_i, there are several additional control signals seen at the UART bus interface driven by the bus master. wb_cyc_i indicates if a valid bus transaction is in progress, wb_stb_i selects the slave, and wb_we_i indicates if the transaction is read or write since the address signal is shared by read and write transactions.

A valid write transaction is marked by we_o, whose assignment is given by the following code:

```
assign we_o =  (wb_we_is & wb_stb_i & wb_cyc_i
    & wbstate==2'b10);
```

The Trojan modifies this assignment to the following:

```
assign we_o =  (wb_we_is & wb_stb_i & wb_cyc_i
    & wbstate==2'b10) | (wb_sel_is == 4'b1001
    & ~re_o);
```

This modification allows registers to be written to even when the slave is not selected or a valid transaction is not occurring, provided a read transaction is not taking place.

Description of (x, $\mathcal{C}$) Pairs: Table III gives the (x, $\mathcal{C}$) pairs analyzed. Similar to the adder coprocessor, the goal is to determine if bus signals can significantly affect the design during conditions in which they are functionally irrelevant. Since the UART is a bus slave, the inputs are the Wishbone address, data, and select signals, which are unspecified when the slave is not selected ($\neg wb\_stb\_i$), or a valid bus cycle is not in progress ($\neg wb\_cyc\_i$). The data and select signals are also unspecified when a read transaction is taking place ($\neg wb\_we\_i$).

Detection Results: The following signals are primary outputs of the UART module: {*wb_dat_o, wb_ack_o, int_o, stx_pad_o, rts_pad_o, dtr_pad_o, baud_o*}. Despite large data-flow graph sizes for several outputs, solving Equation 2 for all primary outputs for each (x, $\mathcal{C}$) pair takes only a few seconds, as seen in Table III. For the Trojan-free design, **there were no false positives**, consistent with our assumption that bus signals should not influence the peripheral while not involved in a valid transaction. Table III also shows that the Trojan circuitry uses wb_adr_i and wb_dat_i in addition to wb_sel_i. This matches the behavior of the Trojan, which allows any data to be written to any register as long as $wb\_sel\_i = 4'b1001$.

C. SMT Solving v. Equivalence Checking

We attempted to use the method presented Section V-B to analyze the UART design using equivalence checking, however ran into an issue resulting from the fact that most of the bus signals, (wb_dat_i, wb_sel_i, etc.) are latched before use in certain portions of the code, whereas the bus signals in the adder coprocessor were never stored.

Conformal is a *combinational* equivalence checking tool, meaning any inputs to storage elements are considered outputs of the combinational logic (pseudo-primary outputs) and all outputs from storage elements are treated as circuit inputs (pseudo-primary inputs). If x is a direct flip-flop input (ex. in data path pipelining), x is a pseudo-primary output during combinational equivalence checking, meaning $f_{x=0}$ and $f_{x=1}$ are trivially non-equivalent. If *only* primary outputs are compared, Trojans using different values of flip-flop input x (and by extension, different values of the stored version of x, x_q) to affect design outputs will not be detected.

The fact that combinational equivalence checking cannot be used to verify sequential circuit behavior is well known, and techniques such as bounded sequential equivalence checking and bounded model checking exist to address this limitation. These techniques use time-frame expansion to capture k cycles of sequential behavior in a purely combinational circuit by copying the original circuit k times and connecting pseudo-primary outputs in one time-frame to pseudo-primary inputs in the following time-frame (ex. [26]).

If the sequential depth, k, of x is known, the method proposed in Section B can analyze a version of the design expanded k time-frames to detect Trojans such as the ones inserted in the UART design. The version of our detection strategy based on SMT solving does not suffer from the same problem because a latch output variable, x_q will always have the corresponding latch input variable, x, in its data-flow graph, meaning any formula for a primary output containing x_q will also contain x.

D. Scalability Issues

Combinational equivalence checking can scale to large designs, but suffers the drawbacks mentioned above. SMT solvers are continually improving, but may not scale to handle an entire SoC. While using formal methods to determine the satisfiability of Equation 1 is the most complete strategy, we would like to note that the problem formulation provided is general enough to develop simulation-based methods from.

One possible simulation-based Trojan detection strategy is to force x to different values when $\mathcal{C}$ occurs during design simulation under the verification tests. Each simulation trace where x is artificially altered can be compared with other traces generated using artificially modified x values as well as the original trace. Any differences in primary output signals or registers seen in the corresponding waveforms is evidence that unspecified functionality may be used to leak information. While incomplete, simulation-based versions of our detection strategy could increase confidence that a large design is Trojan-free.

This strategy can be used at the system integration phase after unspecified functionality in each IP block and subsystem small enough to be analyzed by formal methods is verified by the detection strategy detailed in this work.

VII. Conclusion

We propose a detection methodology for Trojans in unspecified functionality by formulating detection as a satisfiability problem based on the assumption that unspecified functionality should not influence design outputs. Our detection procedure can be followed using a wide variety of tools and techniques, and we give specifics for Trojan detection using 1) data-flow graph analysis and SMT solving, and 2) equivalence checking, meaning our method is applicable to both RT and gate-level designs. We apply our detection methodology to an adder coprocessor and a UART communication controller and successfully and efficiently detect Trojan-infected versions of both designs. The inserted Trojans process information leaked through the bus interface, exploiting the fact that bus protocols only partially specify signal behavior. Our methodology is the first to provide a detection mechanism for this Trojan type.

VIII. Acknowledgments

This work was supported by NSF/SRC STARSS (1526695).

References

[1] S. Adee, "The hunt for the kill switch," *IEEE Spectrum*, vol. 45, no. 5, pp. 34–39, May 2008.

[2] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEEDT*, 2010.

[3] N. Fern, I. San, Ç. K. Koç, and K. T. Cheng, "Hardware trojans in incompletely specified on-chip bus systems," in *DATE*, 2016, pp. 527–530.

[4] A. Waksman *et al.*, "Silencing hardware backdoors," in *Symposium on Security and Privacy*, 2011.

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

[6] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of stealthy malicious logic using boolean functional analysis," in *CCS*, 2013.

[7] D. Sullivan, J. Biggers, G. Zhu, S. Zhang, and Y. Jin, "FIGHT-Metric: Functional identification of gate-level hardware trustworthiness," in *DAC*, 2014.

[8] J. Zhang, F. Yuan, L. Wei, Z. Sun, and Q. Xu, "VeriTrust: Verification for hardware trust," in *DAC*, 2013.

[9] M. Hicks *et al.*, "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," in *Symposium on Security and Privacy*, 2010.

[10] N. Fern, S. Kulkarni, and K.-T. Cheng, "Hardware Trojans hidden in RTL don't cares - Automated insertion and prevention methodologies," in *ITC*, 2015.

[11] N. Fern and K.-T. Cheng, "Detecting hardware trojans in unspecified functionality using mutation testing," in *ICCAD*, 2015, pp. 560–566.

[12] D. Palmer and P. Manna, "An Efficient Algorithm for Identifying Security Relevant Logic and Vulnerabilities in RTL Designs," in *HOST*, June 2013.

[13] T. McComb and L. Wildman, "SIFA: A tool for evaluation of high-grade security devices," in *Information Security and Privacy*. Springer, 2005, pp. 230–241.

[14] X. Li *et al.*, "Secure information flow analysis for hardware design: Using the right abstraction for the job," in *SIGPLAN*, 2010.

[15] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung, "Verifying Information Flow Properties of Firmware using Symbolic Execution," in *DATE*, 2016.

[16] J. Rajendran *et al.*, "Security verification of 3rd party intellectual property cores for information leakage," in *In Proceedings of IEEE VLSI Design*, 2016.

[17] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *VTS*, 2012.

[18] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011.

[19] M. Gario *et al.*, "PySMT: a solver-agnostic library for fast prototyping of smt-based algorithms," 2015.

[20] "Cadence conformal equivalence checker." [Online]. Available: http://www.cadence.com/products/ld/equivalence_checker

[21] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing*, 2015, pp. 451–460.

[22] "Nangate 45nm open cell library," 2011. [Online]. Available: https://www.si2.org/openeda.si2.org/projects/nangatelib

[23] *AMBA AXI and ACE Protocol Specification, Issue E*, ARM, 2013.

[24] "UART 16550 core." [Online]. Available: http://opencores.org/project,uart16550

[25] "Wishbone bus." [Online]. Available: http://opencores.org/opencores,wishbone

[26] F. Lu and K.-T. Cheng, "SEChecker: A sequential equivalence checking framework based on k-th invariants," *IEEE Transactions on VLSI*, vol. 17, no. 6, 2009.