Detecting Malicious Modifications of Data in Third-Party Intellectual Property Cores

Jeyavijayan Rajendran
ECE Department
New York University
Brooklyn, NY, USA
jv.ece@nyu.edu

Vivekananda Vedula
TSR Labs
Austin, TX, USA
vivek.vedula@tsrlabs.com

Ramesh Karri
ECE Department
New York University
Brooklyn, NY, USA
rkarri@nyu.edu

ABSTRACT

Globalization of the system-on-chip (SoC) design flow has created opportunities for rogue elements in the intellectual property (IP) vendor companies to insert malicious circuits (a.k.a. hardware Trojans) into their IPs. We propose to formally verify third party IPs (3PIPs) for unauthorized corruption of critical data such as secret key. Our approach develops properties to identify corruption of critical registers. Furthermore, we describe two attacks where computations can be performed on corrupted data without corrupting the critical register. We develop additional properties to detect such attacks. We validate our technique using Trojans in 8051 and RISC processors and AES designs from Trust-Hub.

1. INTRODUCTION

1.1 Motivation

Fabless System-on-a-Chip (SoC) designers integrate third-party Intellectual Property (3PIP) cores with in-house IP cores to design SoCs. They outsource the fabrication and test phases. 3PIP vendors, foundries, and test companies are distributed worldwide. An SoC designer uses these services to meet the tight time-to-market deadlines and to reduce the design, fabrication, and test costs.

Rogue elements in the 3PIP companies can insert Trojans in their IP [1–3]. The inserted Trojans may be conditionally triggered or always on. When triggered, a Trojan may result in a deadlock or failure of the system (overt attack), or create a backdoor allowing the attacker to gain remote access to the system (covert attack) [2, 3].

To build a trustworthy SoC design, it is necessary to ensure the trustworthiness of the 3PIPs. However, since this is not always possible, the SoC integrator should ensure that all the security vulnerabilities in any of the 3PIPs are detected or their effects muted before they damage the system.

1.2 Previous work

Trojan-detection techniques in 3PIPs can be broadly classified into code/structural analysis and formal verification techniques.

Code/structural analysis techniques. Since 3PIPs are typically delivered as Register Transfer Level (RTL) VHDL/Verilog codes, code coverage analysis is performed on RTL codes to identify suspicious signals that may be a part of a Trojan [4, 5]. Even 100% coverage of the RTL code in a design does not guarantee that it is fault-free [6]. Hence, code coverage analysis does not guarantee its trustworthiness.

DAC '15, June 07 - 11, 2015, San Francisco, CA, USA Copyright is held by the owner/author(s). Publication rights licensed to ACM.
http://dx.doi.org/10.1145/2744769.2744823
Copyright 2015 ACM ACM 978-1-4503-3520-1/15/06 ...$15.00.

Alternately, an SoC integrator may automatically analyze the 3PIP code and mark suspicious signals using controllability and reachability values of signals [7]. FANCI marks gates with low activation probability as suspicious [8]. VeriTrust marks gates that are not driven by functional inputs as suspicious [9]. The implicit assumption here is that those gates are driven by Trojans, as they do not perform any computation on functional inputs. The SoC integrator then manually analyzes the small number of suspicious gates to determine if they are part of a Trojan.

DeTrust exploits the limitations of FANCI and VeriTrust to design Trojans that bypass them [10]. To bypass FANCI, DeTrust designs Trojans whose trigger vector arrives over multiple clock cycles. If the probability of activating a signal is below a pre-determined threshold, FANCI marks it as suspicious. For example, if a 128-bit trigger arrives in one clock cycle, the probability of activating the trigger signal is 2^{-128}, and FANCI marks it as suspicious. However, DeTrust makes the trigger signals arrive as four-bit nibbles over 32 clock cycles. Now, FANCI computes the probability of activating the trigger signal to be 2^{-4}. Since this value is significantly higher, FANCI does not mark this signal as suspicious. To bypass VeriTrust, DeTrust ensures that each gate in the Trojan is driven by a subset of functional inputs.

The limitations of code/structural analysis techniques are: (1) they do not guarantee Trojan detection [10], (2) they burden the designer with manual analysis, and (3) they analyze only the combinational parts of the design.

Formal verification techniques. An SoC integrator and a 3PIP vendor can agree upon a pre-defined set of security properties that the IP should satisfy [11]. The SoC integrator can check the 3PIP for these properties. To check if a design honors these properties, one converts the target design into a proof checking format (for example, Coq). This technique has been demonstrated to detect data leakage [12] and malicious modifications to registers [13].

The limitations of this technique are: (1) One can check if a design satisfies pre-defined properties, but not if the design has additional vulnerabilities while satisfying these properties [13]. (2) Lack of automation to convert VHDL/Verilog to Coq format. (3) The VHDL/Verilog and Coq representations of the target design may not be equivalent; If a Coq representation of a design is considered trustworthy, it does not necessarily mean that the corresponding VHDL/Verilog representation is trustworthy.

Jasper uses a proprietary "taint propagation" technology to identify if secret data can be leaked by exploiting design bugs [14]. However, it does not target Trojans. The authors of [15] identified and verified three security properties of SoCs using taint-propagation techniques: (1) the firmware should not read the hardware encryption keys, (2) the firmware should not modify the access privileges of memory and input/outputs, and (3) the host software should not modify the SoC security monitor. However, these techniques target unintentional design bugs, but not Trojans.

1.3 Contribution: Formally verifying 3PIPs for data-corrupting Trojans

We propose to detect Trojans in 3PIP that corrupt regis-

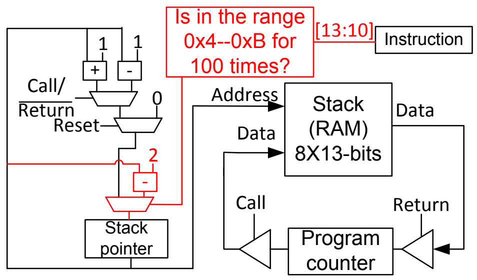

Figure 1: A Trojan corrupting the stack pointer of a RISC processor [16, 17]. The valid ways to update the stack pointer are: (1) a CALL instruction increments its value by 1, (2) a RET instruction decrements its value by 1, and (3) a RESET instruction sets its value to 0. The Trojan is triggered when the bits [13:10] of the instruction register are in the range 0x4–0xB for 100 times. The Trojan decrements the stack pointer value by 2. Trojan components are shown in red.

ters that store critical data such as a secret key of a cryptographic design, a stack pointer of a processor, or a destination address register of a router. We use bounded model checking (BMC) to detect Trojans. The input to the model checker is the target "no-data-corruption" property and a formal description of the design in temporal logic[1]. The output of the model checker is a set of states which satisfy the given "no-data-corruption" property or a witness (sequence of inputs) which violates the property. The property that we check is "does critical information get corrupted?" For instance, we check if the stack pointer in Figure 1 can be corrupted.

The advantages of formally verifying a design for security properties are: (1) any design can be verified, (2) it guarantees detection of data-corrupting Trojans and produces the trigger condition for the Trojan, and (3) it can be used in conjunction with other property-based Trojan detection techniques [11–13].

Unlike code/structural analysis, it (1) detects all Trojans that corrupt data, (2) is automated (no need of manual analysis), and (3) applies to both combinational and sequential parts of the design. Unlike proof carrying code, our technique detects Trojans that do not violate the pre-defined properties and is automated.

1.4 Motivational example

A Trojan which corrupts the critical data stored in a register, R, is formally defined as

$$\exists \;\; i_{trigger} \in I, i_{trigger} \notin \mathbb{V} \ni D \models (R_{t-1} \neq R_t) \qquad (1)$$

where I is the set of possible input patterns across all clock cycles $[1, t]$, and R_t is the value of the register R at clock cycle t. D is the design and $\mathbb{V}$ is the set of valid ways one can update R. On applying $i_{trigger}$, R is corrupted. The register that holds the critical data is called the *critical register*. This critical register can be a key register in a cryptographic design, a stack pointer in a processor, or a register that holds destination address in a router.

Example 1: In Figure 1, the stack pointer of a RISC processor is corrupted when bits 13-10 of the instruction are in the range 0x4–0xB for 100 times. When triggered, the Trojan decrements the stack pointer by two, which can lead to control-flow attacks.

Based on the trigger, a Trojan can be (i) always on (i.e. no trigger), (ii) triggered by only current inputs, (iii) triggered after a specific number of clock cycles, and (iv) triggered by inputs arriving over multiple clock cycles [2,3]. FANCI and VeriTrust deal with Trojans of type (i) and (ii). DeTrust designed Trojans of type (iii) and (iv) to defeat them. Our technique can detect data-corrupting Trojans of all types.

1.5 Organization of the paper

The rest of the paper is organized as follows: Section II details the threat model, prior work, and background. Section III derives a simple property to detect Trojans that corrupt critical data. Section IV describes how this simple property can be violated and develop additional properties to detect such violations. Section V concludes the paper.

2. THREAT MODEL AND BACKGROUND

2.1 Threat model

A 3PIP vendor or a rogue element in the 3PIP vendor company is the **attacker**. The attacker seeks to subvert the security of the SoC that is using his IP. He introduces hardware Trojans in the IP to corrupt critical data. He only inserts Trojans whose trigger and payload characteristics are "digital." He cannot design a Trojan that depends on the physical characteristics of the SoC as these characteristics are determined by the design-synthesis constraints; the 3PIP vendor has no control over the design constraints imposed on the SoC by the SoC integrator. If a 3PIP has "non-volatile" components, a designer can identify them as he needs to manufacture them; thus, we consider Trojans that do no use non-volatile components.

The SoC integrator is the **defender**. His objective is to detect Trojans, if any, in the 3PIP. We assume that the defender has access to the RTL/gate-level netlist of the 3PIP and hence can verify its function. For example, he can verify that the value in the stack pointer in Figure 1 always increments by 1 on a CALL instruction and decrements by 1 on a RET instruction. Furthermore, he knows the functionality of the input and output ports of the 3PIP from the specification.

Protocol to verify trustworthiness of an IP. We follow the protocol outlined in [11–14]. The SoC integrator and the IP vendor agree upon a set of security properties for the design. For instance, one property can check for the valid ways to update a stack pointer in a processor. An SoC integrator performs functional verification. The attacker in the IP design house can insert Trojans that corrupt critical data, while satisfying the agreed upon security properties, and passes functional verification. The task of the SoC integrator is two-fold: (1) check if the design satisfies the agreed upon security properties, and (2) check if the design has Trojans that corrupt data without violating these properties.

2.2 Formal methods in hardware design

Formal verification is an approach to ensure that safety-critical components in a design are exhaustively tested for correctness [18]. Quality criteria may be specified using properties described in temporal logic and its variations [19]. In linear time temporal logic (LTL), the notion of time is that of a linearly ordered set (this can be thought of as a possible sequence of states).

Model checking is the process of analyzing a design for the validity of properties stated in temporal logic. A model checker takes the Verilog code along with the property written as a Verilog assertion and derives a Boolean satifiability (SAT) formulation for validating/invalidating the property. This SAT formulation is fed to a SAT engine, which then searches for an input assignment that violates the property [20].

Bounded model checking. In practice, designers know the bounds on the number of steps (clock cycles) within which a property should hold. In Bounded Model Checking (BMC), a property is determined to hold for at least a finite sequence of state transitions. The Boolean formula for validating/invalidating the target property is given to a SAT engine, and if a satisfying assignment is observed within T clock cycles, that assignment is a witness against the target property [21]. We develop properties to detect Trojans that corrupt critical data and verify the target design for satis-

[1]Temporal logic is a representation of a design as a sequence of states.

faction of these properties using a bounded model checker.

2.3 Formal methods in hardware security

SAT-based techniques can be used to detect fault attacks [22]. Satisfiability Modulo Theory-based techniques can evaluate the strength of software countermeasures against side-channel attacks [23]. These techniques do not target Trojans.

3. DETECTING DATA CORRUPTION

3.1 Data corruption

Consider a Trojan corrupting the data stored in an N-bit register, R. The input pattern i applied at clock cycle t is denoted as i_t. Let I be the set of all possible input patterns and $i \in I$. Let S be the sequence of inputs $i_1, i_2, ..., i_T$ applied in the interval $[1, T]$. Let $\mathbb{V}$ be the set of valid ways one can update R.

$$\bigwedge_{x=1}^{N} \forall (S \notin \mathbb{V}) \Rightarrow R_{x,t-1} = R_{x,t} \tag{2}$$

where, $R_{x,t}$ is the value of the xth bit of R at clock cycle t. This property can be written as a Verilog assertion and given to a BMC tool along with the target design. The bound for BMC is set as T clock cycles. If BMC does not detect a counterexample that violates this property within T clock cycles, there is no input sequence, other than those in $\mathbb{V}$, that modify R. However, if BMC outputs a counterexample that violates this property, R can be corrupted.

Example 2: Consider the stack pointer of a RISC processor shown in Figure 1. On checking the corruption of the stack pointer using the property in Equation (2), BMC produces a counterexample, which has 100 ADD instructions. Note that an ADD instruction should not affect the stack pointer in a Trojan-free design.

Partial corruption: Even if an attacker modifies a subset of the bits in R, the property can detect this Trojan as it checks for individual bits of R.

Multi-cycle triggers: It accounts for triggers arriving at a single clock cycle or over multiple clock cycles.

Example 3: Consider the Trojan in AES-T800 from the Trust-hub [16][2]. The Trojan corrupts the least-significant eight bits of the secret key. One needs to apply four pre-selected plaintexts consecutively to trigger the Trojan. The Trojan will not be triggered if these plaintexts do not arrive in sequence. BMC produces a counterexample for the no-data-corruption property (Equation (2)), consisting of the four plaintexts which trigger the Trojan.

3.2 Speeding-up using an ATPG

Motivation. The time complexity of BMC increases with an increase in the number of clock cycles. Hence, one can evaluate the trustworthiness for the first T clock cycles, where T is the maximum number of clock cycles for which BMC is performed. For clock cycles $\geq T$, one cannot guarantee the trustworthiness of the design. This is similar to the trustworthiness guarantees provided in [24]. One then needs to reset the design after every T clock cycles. Resetting registers in processors while reducing the overhead is described in [24].

Example 4: In Figure 1, if the design is unrolled for 400 clock cycles, BMC can find a set of instructions that trigger the Trojan. However, if the design is unrolled for less than 400 clock cycles, BMC does not produce any counterexample. Hence, one needs to check for as many clock cycles as possible.

One can use automatic test pattern generation (ATPG) to ensure the trustworthiness over a large number of clock cycles [25][3]. The property is modeled as a monitor circuit, which is appended with the target circuit [26]. One can generate a test for a stuck-at-1 fault at the output of this monitor circuit so as to force the ATPG to generate an input pattern that violates this property. If an ATPG generates a test pattern (counterexample), the property is violated. If the fault is not detected, then the property holds true. If an ATPG returns untestable, we cannot guarantee the trustworthiness of the design. The monitor circuit is needed only for validation and is not implemented in silicon.

3.3 Results

3.3.1 Experimental setup

We generated Verilog assertions for the data corruption property in Section 3.1 for the designs in the Trust-Hub benchmark suite [16]. We used Trojans designed by DeTrust [10]. The valid ways to update the registers in the benchmark designs were obtained from their datasheet. These assertions were embedded into the respective designs and provided as input to the BMC engine of the SMV tool from Cadence [20]. We used the ATPG tool from Synopsys Tetramax with the "full-sequential" option [27]. We used an Intel(R) Xeon E5-2450L 32 cores CPU with 128GB memory operating at 1.80GHz to run the simulations. The Trojans from the Trust-Hub benchmark suite do not violate the functional specification of the design until they are triggered; thus, Trojan-infected designs do not violate conventional functional properties and remain undetected.

The first three columns in Table 1 show the characteristics of the Trojans. In case of MC8051 and RISC, the Trojans corrupt the critical registers of these processors. In case of AES, they corrupt the secret key register. The trigger condition varies from a trigger arriving in one clock cycle (e.g., MC8051-T700) to a trigger arriving over multiple clock cycles (e.g., MC8051-T400) to a trigger arriving after a specific number of clock cycles (e.g., AES-T1200).

3.3.2 Detection capability

A design is infected with a data-corrupting Trojan even if one bit of the critical register is corrupted. Columns 4-6 and 10 in Table 1 show the detection capability of different techniques. FANCI [8] and VeriTrust [9] did not detect any of these Trojans as reported by DeTrust [10]. Our technique (both BMC- and ATPG-based) detected all Trojans except for AES-T1200.

In AES-T1200, the Trojan is triggered after $2^{128} - 1$ clock cycles. The Trojan consists of a 128-bit counter to count the number of clock cycles. To generate a counterexample that triggers this Trojan, one needs to unroll the design for $2^{128} - 1$ clock cycles, which is computationally infeasible. When BMC is performed for 2^{14} clock cycles, the key register is not corrupted. Consequently, our technique concluded this design is trustworthy for 2^{14} clock cycles.

The technique is oblivious to the structure of the Trojan. For example, the Trojan in MC8051-T400 has a finite state machine, and the Trojan in MC8051-T800 has only combinational logic. Our technique (both BMC- and ATPG-based) detected both these Trojans by selecting instructions that trigger them. The technique is independent of the underlying design; it detected Trojans in processors (MC8051 and RISC) and crypto modules (AES).

Columns 7 and 8 in Table 1 show the memory usage and the time taken for our property to detect Trojans in BMC implementation. The memory usage for BMC is in the range of GBs because BMC makes multiple copies of the design for the number of clock cycles unrolled. Furthermore, all the Trojans were detected within 100 seconds. Columns 11 and 12 in Table 1 show the memory usage and the time taken

[2]The payload of Trojan is modified to corrupt instead of leaking the key.

[3]ATPG is faster and more efficient than a SAT-based BMC because it efficiently balances depth-first and breadth-first searches to generate a counterexample or its absence, while BMC uses a depth-first search [25].

Table 1: Detecting the Trojans from the Trust-Hub [16]. We used Trojan structures from DeTrust [10]. The number in the parentheses in the trigger condition column indicates the clock cycle in which the trigger arrives. "N/A" indicates that no counterexample was found to violate the property. We ran the tools for 100 seconds to obtain the maximum number of clock cycles for which a design can be unrolled.

Trojan			FANCI	VeriTrust	BMC				ATPG			
Name	Trigger condition	Payload	Detected?	Detected?	Detected?	Time (s)	Memory (GB)	Max.# of clk cycles	Detected?	Time (s)	Memory (GB)	Max.# of clk cycles
MC8051-T400	Instruction = (1) MOV A, Data (2) MOVX A, @R1 (3) MOVX A, @DPTR (4) MOVX @R1, A	Prevents interrupt	No	No	Yes	0.01	0.01	140	Yes	0.001	0.01	480
MC8051-T700	MOV A, Data	Modifies the data to 0x00	No	No	Yes	0.01	0.09	160	Yes	0.001	0.01	450
MC8051-T800	Input data of UART =0xFF	Decrements stack pointer by two	No	No	Yes	0.01	0.1	160	Yes	0.001	0.01	430
RISC-T100	After 100 instructions	Increments program counter by two	No	No	Yes	73.86	1.2	110	Yes	8.57	0.2	360
RISC-T300	whose 4 MSBs	Modifies the data written to memory	No	No	Yes	75.63	3.2	120	Yes	0.56	0.19	360
RISC-T400	are in the range 0x4–0xB	Modifies the data address to 0x00	No	No	Yes	74.67	1.8	120	Yes	0.58	0.19	360
AES-T700	Plaintext = 128'h00112233445566 778899aabbccddeeff	Modifies LSB 8-bits of key register	No	No	Yes	40.6	2.01	50	Yes	0.22	0.74	150
AES-T800	Plaintext = (1) 128'h3243f6a8885a3 08d313198a2e0370734 (2) 128'h0011223344556 6778899aabbccddeeff (3) 128'h0 (4) 128'h1	Modifies key register	No	No	Yes	40.6	2.01	70	Yes	0.41	0.77	160
AES-T1200	After 2^{128} clock cycles	Modifies key register	No	No	N/A	N/A	N/A	160	N/A	N/A	N/A	160

for ATPG implementation. The memory usage of ATPG is an order of magnitude lower than that of BMC. In this case, all the Trojans were detected within a second.

Table 2: Valid ways to update registers in RISC [17]

Register	**Cycle**	**Valid way**	**Value**
Program counter	Any	Reset=1	0x00
	4	Stall=0	Increment by 1
	4	Interrupt=1 & Stall=0	0x04
	4	Return =1 & Stall=0	Stack array[Stack pointer]
	4	Goto =1 & Stall =0	{PC Latch,Instr. register}
	4	Destination = PCL	{PC Latch,Output of ALU}
Stack pointer	Any	Reset=1	0x00
	2	Return=1	Decrement by 1
	4	Call=1	Increment by 1
Interrupt enable	Any	Extl. interrupt	0x01
	Any	Overflow	0x01
	Any	Write complete	0x01
EEPROM data	4	Stall=0 & EEPROM read =1	EEPROM input
EEPROM address	4	Stall=0	RAM[0x09]
Instruction register	4	–	RAM[Program counter]
Sleep flag	Any	Reset=1	0
	4	Sleep inst.	1

Columns 9 and 13 in Table 1 show the maximum number of clock cycles for which BMC and ATPG, respectively, can unroll a design. For this experiment, we ran the tools for 100 seconds. The number of clock cycles unrolled by ATPG is 3× that of BMC. All these designs were unrolled for several thousand clock cycles, when there is no time constraint set for the tool. To be prudent, the SoC integrator has to reset the design once the number of clock cycles exceeds this value. Since we unrolled the design for several thousand clock cycles, the integrator needs to reset it only once every several thousand clock cycles.

Checking for data corruption did not result in any false negatives as it detected all the Trojans. To check for false positives, we checked for data corruption in Trojan-free MC8051, RISC, and AES designs from Trust-Hub. Our technique did not flag these designs as infected with data-corrupting Trojans.

3.4 Case Study: Data corruption in a RISC processor

The RISC processor in the Trust-hub benchmark suite is a 4-cycle non-pipelined processor [16]. The valid ways to update the registers were obtained from its datasheet [17]. Table 2 shows some of the critical registers in this design and the valid ways to update them.

The RISC-T100 Trojan affects the program counter when triggered. When there is no stall, the program counter value increments by 1 If there is a "Goto" instruction, the instruction register values are copied to the program counter. Other instructions should not affect the program counter. The Trojan is triggered when the four MSBs values of the instruction register are in the range 0x4–0xB for 100 clock cycles. When triggered, the program counter value is incremented by two. Our technique (both BMC- and ATPG-based) detected this Trojan because non-Goto instructions changed the program counter.

The RISC-T300 Trojan affects the $EEPROM\ data$ register when triggered. When there is no stall, the values in $EEPROM\ input$ register are copied to $EEPROM\ data$ register when $EEPROM\ read$ signal is enabled. The triggering mechanism is same that of RISC-T100. Our technique (both BMC- and ATPG-based) detected this Trojan because the contents of $EEPROM\ data$ register are changed when $EEPROM\ read$ signal is disabled.

The RISC-T400 Trojan changes the $EEPROM\ address$ register to 0x00 when triggered. When there is no stall, the values in the special purpose register, RAM[0x09], are copied to $EEPROM\ address$ register. The triggering mechanism is same that of RISC-T100. Our technique (both BMC- and ATPG-based) detected this Trojan because the contents of $EEPROM\ address$ register were changed when there was a stall.

4. BEYOND CHECKING CRITICAL REGISTERS

While one can check if all registers (not necessarily critical) in a design for corruption, it is practically not possible to list the valid ways to update/modify for every register, as there can be thousands of registers in a design. Thus, an SoC integrator checks only the registers that are deemed critical to him. Hence, a rogue IP designer can use the other registers in the design to corrupt critical data while not corrupting the critical register. In this section, we describe two attacks where computations can be performed on corrupted data without corrupting the critical register.

4.1 Attack 1 – Use pseudo-critical registers

A malicious 3PIP vendor or a rogue element in a 3PIP design house can deceive the defender by introducing registers, whose input is the output of a critical register. Such a register is called a *pseudo-critical register.* It feeds the fan-out logic of the critical register, instead of the critical register. An attacker, instead of using the contents of the critical registers, corrupts the contents of pseudo-critical registers and uses them. The defender checks the critical register for corruption, but not the pseudo-critical register. Consequently, this Trojan is not detected.

Example 5: Consider the modified stack pointer of the RISC processor shown in Figure 2. The *stack pointer* holds the critical data, and the defender checks if it can be corrupted. However, the stack pointer feeds the pseudo-critical stack pointer. The contents of the pseudo-critical stack pointer are corrupted when the Trojan is triggered.

Defense. One can detect pseudo-critical registers as follows: Let R be a critical register. P is a pseudo-critical register if every update of R updates P. Formally,

$$\forall(S \in \mathbb{V}) \Rightarrow \bigvee_{x=1}^{N} (P_{x,t} = R_{x,t-1}) \vee (P_{x,t} = \neg R_{x,t-1}) \quad (3)$$

Example 6: In Figure 2, the SoC integrator identifies the pseudo-critical stack pointer by setting it as P and the stack pointer as S in Equation (3). On checking it for the property in Equation (2), he determines that it can be corrupted.

It is not necessary that the contents of R are directly copied to P. Sometimes, a Boolean function of the contents of R can be copied to P. For example, an attacker can invert the content of R before copying it to P. The output of P can be inverted once again to retain the original value.

For an N-bit register, there are 2^{2^N} possible Boolean functions. Since Equation (3) deals with individual bits of R, there are only four possible Boolean functions for a bit x: $\{1, 0, x, \neg x\}$. An attacker cannot force a bit in P to constant 1 or 0, as it is equivalent to a stuck-at fault; such faults are revealed during functional testing with valid ways, $\mathbb{V}$, that update R and thereby P. Hence, he can force a bit in P to be either x or $\neg x$. Equation (3) checks for both these cases. A pseudo-critical register can also be before the critical register. In such cases, one needs to use $P_{x,t-1}$ and $R_{x,t}$, instead of $P_{x,t}$ and $R_{x,t-1}$, respectively, in Equation (3).

Any register in the design can be a pseudo-critical register. Hence, a defender has to check for all pairwise combinations of registers using Equation (3). Identifying a pseudo-critical register does not necessarily mean it can be corrupted. A designer has to check if its contents can be corrupted using Equation (2).

4.2 Attack 2 – Use bypass registers

An attacker can bypass the output of a critical register using a *bypass register* as shown in Figure 3. The bypass register holds a corrupted value. Once the Trojan is triggered, the multiplexer at the fan-out logic of the critical register selects the corrupted value from the bypass register instead of the valid value from the critical register. The attacker evades the detection technique as it checks the critical register and not the bypass register.

Example 7: Consider the modified stack pointer of the RISC processor in Figure 3. An attacker informs that stack pointer holds the critical data. The defender checks only the stack pointer. The bypass stack pointer holds the corrupted data. When the Trojan is triggered, the contents of the bypass stack pointer is selected instead of the original stack pointer.

Defense. When the Trojan is triggered, the output of the critical register is ignored, and only that of the bypass register is used. Consequently, the critical register does not have any effect on the output. This is not possible unless the critical register is bypassed. We can use this property to detect bypass registers. Let O be the set of output ports and R be the critical register. $o_{t+1,p}$ is the value of output port o at clock cycle $t+1$ when the value of R is p. Formally,

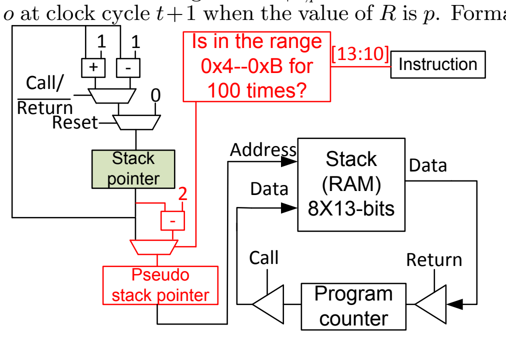

Figure 2: Pseudo-critical registers: The *stack pointer* (green) holds the critical data, and the defender checks if it can be corrupted. However, the stack pointer feeds the pseudo-critical stack pointer. The contents of the pseudo-critical stack pointer are corrupted when the Trojan is triggered. Trojan components are shown in red.

```
Input   : Netlist,ListofRegisters,ListofCriticalRegisters,ValidWays
D = Netlist;
For each Register_i in ListofCriticalRegisters do
    V = ValidWays[Register_i];
    For each Register_j in ListofRegisters do
        if CheckPseudoCritical(D,Register_i,Register_j,V,T) then
            ListofCriticalRegisters +=Register_j;
        end
    end
    (TrojanFlag, CounterExample) =
            CheckForCorruption(D,Register_i,V,T);
    if TrojanFlag == 1 then
        print("Register_i is corrupted for CounterExample");
        exit();
    end
    (TrojanFlag, CounterExample) =
    CheckBypass(D,Register_i,V,T);
    if TrojanFlag == 1 then
        print("Register_i is bypassed for CounterExample");
        exit();
    end
end
print("No data-corruption Trojan found for T clock cycles");
```

Algorithm 1: Detecting data corruption

$$\neg\exists S \;\; \forall i_{(t+1)} \in I \;\; \forall p, q \in \mathbb{R}, p \neq q \Rightarrow \bigwedge_{o \in O} o_{t+1,p} = o_{t+1,q} \quad (4)$$

where $\mathbb{R}$ is the set of all possible values of R. If the latency between the critical register and o is L, one needs to check o_t at $t + L^{th}$ clock cycle.

Example 8: In Figure 3, the critical register is the stack pointer. The program counter should be loaded with different values when the stack pointer is changed. When the Trojan is triggered, the value in the bypass stack pointer is used. Thus, the contents of the original stack pointer will not have any effect on program counter. This is not true for the RISC processor. Consequently, Equation (4) is violated.

4.3 Putting it all together

Algorithm 1 shows the steps to detect Trojans that corrupt critical registers even in the presence of pseudo-critical and bypass registers. Its inputs are the netlist of the design (D), the list of registers, the list of critical registers, the set of valid ways to update them, and the number of clock cycles (T) for which the trustworthiness has to be evaluated. A defender can obtain the list of registers by parsing the netlist. First, the algorithm identifies pseudo-critical registers for each critical register and adds them to the list of critical registers. Second, the algorithm checks if a critical register can be corrupted or not. If it can be corrupted, the algorithm exits by stating "Trojan found." Additionally, it presents a counterexample (i.e. a set of input sequences) that corrupts the register. Finally, it also finds bypass registers, if any. If the algorithm does not find any input sequences to corrupt the critical registers or bypass registers, it terminates. The algorithm ensures the trustworthiness of the design only for the specified number of clock cycles (T).

If a Trojan corrupts a critical register or bypasses it, there has to be a circuit path between the Trojan and the critical register. Thus, the Trojan circuit can be either at the front, back, or can be converging with the critical register in its fan-out logic. Equations (2) and (3) detect Trojan circuits which

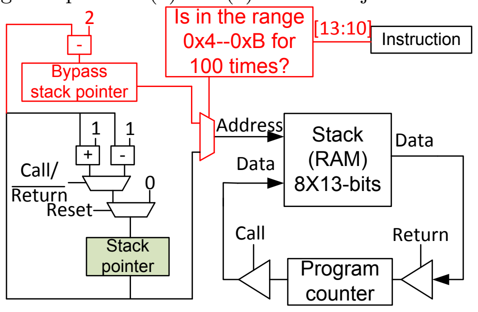

Figure 3: Bypass registers: If according to the IP vendor the stack pointer (green) holds the critical data, the defender checks if it can be corrupted. Now, if an attacker inserts a Trojan, which when triggered, selects the output of the bypass stack pointer instead of the original stack pointer.

Table 3: Detecting pseudo-critical and bypass registers. We used Trojan structures from DeTrust [10]. "N/A" indicates that a counterexample was not found to violate the property. We ran the tools for 100 seconds for the tools to obtain the maximum number of clock cycles for which a design can be unrolled.

Name	Detected?				Max. # of clock cycles			
	FAN-CI [8]	Veri-Trust [9]	BMC	ATPG	Pseudo-critical		Bypass	
					BMC	ATPG	BMC	ATPG
MC8051-T400	No	No	Yes	Yes	160	450	180	480
MC8051-700	No	No	Yes	Yes	110	500	120	540
MC8051-T800	No	No	Yes	Yes	140	500	150	540
RISC-T100	No	No	Yes	Yes	120	290	130	190
RISC-T300	No	No	Yes	Yes	130	280	150	190
RISC-T400	No	No	Yes	Yes	130	290	150	130
AES-T700	No	No	Yes	Yes	60	150	45	120
AES-T800	No	No	Yes	Yes	70	150	50	130
AES-T1200	No	No	N/A	N/A	70	150	50	110

are at front or back of the critical register. Equations (2) and (4) detect Trojan circuits which converge with the critical register. Thus, all Trojans that corrupt critical registers or bypass them can be detected.

4.4 Results

To analyze the effectiveness of the techniques in detecting pseudo-critical and bypass registers, we modified the designs in the Trust-hub benchmark suite to include such registers. We used Trojans from DeTrust [10]. For this experiment, we ran the tools for 100 seconds.

Table 3 summarizes the results. FANCI [8] and VeriTrust [9] cannot detect any of the Trojans as mentioned by DeTrust [10]. Our techniques (both BMC and ATPG-based) detected pseudo-critical and bypass registers in all designs except AES-T1200. In AES-T1200, the bypass register is activated only after 2^{128} clock cycles. Our technique flags this design as "trustworthy" for the 2^{14} clock cycles because the critical register is not bypassed till then.

It is easier to check the property for pseudo-critical register, when a register is easily controllable; It is easier to check the property for bypass register, when a register is easily observable. The critical register (key register) of AES is closer to the inputs. Thus, it is relatively easier to control than observe. Consequently, the number of clock cycles unrolled to check for pseudo-critical registers is greater than that of bypass registers. Contrarily, the critical registers in MC8051 and RISC are closer to the outputs. Thus, they are relatively easier to observe than control. Consequently, the number of clock cycles unrolled to check for pseudo-critical registers is less than that of bypass registers.

Similar to Table 1, the number of clock cycles unrolled by ATPG is 2.5× than that by BMC. Given sufficient time (30 minutes), all designs were unrolled for >1000 clock cycles.

4.5 Limitations

4.5.1 *Detecting Trojans with one way functions*

In a one-way function (OWF), it is computationally infeasible to determine an input pattern for an output pattern (for example, cryptographic hash functions). If an attacker uses such OWFs to trigger the data-corrupting Trojan, our technique cannot generate a counterexample; A BMC tool or an ATPG tool exits by stating the design is untestable. Hence, we cannot verify the trustworthiness of such designs. However, OWFs are expensive to build in hardware. For example, SHA-1, an OWF, requires 60K logic gates.

4.5.2 *Scalability*

BMC and ATPG are NP-complete problems [20,27]. However, efficient heuristics developed for practical circuits reduce this complexity of ATPG to polynomial in the number of gates in the circuit [28]. Furthermore, companies regularly use BMC and ATPG techniques on large-scale designs [20]. Thus, these techniques are practical and scalable. We have demonstrated how they can be repurposed for security.

5. CONCLUSION

Our technique detected Trojans designed by DeTrust, which bypass two state-of-the-art techniques FANCI and VeriTrust [10]. In this work, we targeted Trojans that modify data. Trojans can also modify functionality, leak data, or reduce reliability/performance. By applying the no-data-corruption properties on the registers of the controllers in a design, one can ensure that the system always is in a valid state; thereby, one can potentially detect Trojans that modify functionality. One can develop a similar set of properties to detect data leakage. While we developed properties to detect Trojans, one can reuse them to detect security vulnerabilities caused by design bugs and for security validation. Furthermore, the developed properties need to adapted for the target processor.

6. REFERENCES

[1] "Defense Science Board (DSB) study on High Performance Microchip Supply," http://www.acq.osd.mil/dsb/reports/ADA435563.pdf, 2005.

[2] S. Bhunia, M. Hsiao, M. Banga, and S. Narasimhan, "Hardware Trojan Attacks: Threat Analysis and Countermeasures," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1229–1247, 2014.

[3] M. Tehranipoor and F. Koushanfar, "A Survey of Hardware Trojan Taxonomy and Detection," *IEEE Design and Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.

[4] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware Trojans in third-party digital IP cores," *IEEE Intentional Symposium on Hardware Oriented Security and Trust*, pp. 67–70, 2011.

[5] M. Banga and M. Hsiao, "Trusted RTL: Trojan detection methodology in pre-silicon designs," *IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 56–59, 2010.

[6] J. Jou and C. J. Liu, "Coverage analysis techniques for HDL design validation," *IEEE Asia Pacific Conference on Chip Design Languages*, 1999.

[7] H. Salmani and M. Tehranipoor, "Analyzing circuit vulnerability to hardware Trojan insertion at the behavioral level," *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, pp. 190–195, 2013.

[8] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis," *ACM Conference on Computer and Communications Security*, pp. 697–708, 2013.

[9] J. Zhang, F. Yuan, L. Wei, Z. Sun, and Q. Xu, "VeriTrust: Verification for hardware trust," *IEEE/ACM Design Automation Conference*, pp. 1–8, 2013.

[10] J. Zhang, F. Yuan, and Q. Xu, "DeTrust: Defeating Hardware Trust Verification with Stealthy Implicitly-Triggered Hardware Trojans," *ACM Conference on Computer and Communications Security*, pp. 153–166, 2014.

[11] E. Love, Y. Jin, and Y. Makris, "Proof-Carrying Hardware Intellectual Property: A Pathway to Trusted Module Acquisition," *IEEE Trans. on Information Forensics and Security*, vol. 7, no. 1, pp. 25–40, 2012.

[12] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," *IEEE VLSI Test Symposium*, pp. 252–257, 2012.

[13] ——, "A proof-carrying based framework for trusted microprocessor IP," *IEEE/ACM International Conference on Computer-Aided Design*, pp. 824–829, 2013.

[14] Jasper, "JasperGold: Security Path Verification App," http://www.jasper-da.com/products/jaspergold-apps/security_path_verification_app, 2014.

[15] P. Subramanyan and D. Arora, "Formal verification of taint-propagation security properties in a commercial SoC design," *Design, Automation and Test in Europe Conference and Exhibition*, pp. 1–2, 2014.

[16] M. Tehranipoor, R. Karri, F. Koushanfar, and M. Potkonjak, "Trusthub," *http://trust-hub.org*.

[17] Microchip Technology, "PIC16F84A Data sheet," ww1.microchip.com/downloads/en/DeviceDoc/35007b.pdf, 2001.

[18] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal Methods: Practice and Experience," *ACM Computing Surveys*, vol. 41, no. 4, pp. 19:1–19:36, 2009.

[19] A. Pnueli, "The temporal semantics of concurrent programs," *Semantics of Concurrent Computation*, vol. 70, pp. 1–20, 1979.

[20] "Cadence: Smv," http://www.cadence.com/products/fv/pages/default.aspx, 2005.

[21] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1579, pp. 193–207, 1999.

[22] L. Feiten, M. Sauer, T. Schubert, A. Czutro, E. Bohl, I. Polian, and B. Becker, "#SAT-based vulnerability analysis of security components — A Case Study," *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, pp. 49–54, 2012.

[23] H. Eldib, C. Wang, and P. Schaumont, "SMT-Based Verification of Software Countermeasures against Side-Channel Attacks," *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 62–77, 2014.

[24] A. Waksman and S. Sethumadhavan, "Silencing Hardware Backdoors," *IEEE Symposium on Security and Privacy*, pp. 49–63, 2011.

[25] V. Boppana, S. Rajan, K. Takayama, and M. Fujita, "Model Checking Based on Sequential ATPG," *Computer Aided Verification*, vol. 1633, pp. 418–430, 1999.

[26] J. Abraham and V. Vedula, "Verifying properties using sequential ATPG [IC design]," *International Test Conference*, pp. 194–202, 2002.

[27] Synopsys, "Tetramax ATPG," http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Test/Pages/TetraMAXATPG.aspx, 2014.

[28] M. Prasad, P. Chong, and K. Keutzer, "Why is ATPG easy?" *IEEE/ACM Design Automation Conference*, pp. 22–28, 1999.