Detecting Hardware Trojans in Unspecified Functionality Using Mutation Testing

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

Abstract—**Existing functional Trojan detection methodologies assume Trojans violate the design specification under carefully crafted rare triggering conditions. We present a new type of Trojan that leaks secret information from the design by only modifying *unspecified functionality*, meaning the Trojan is no longer restricted to being active only under rare conditions. We provide a method based on mutation testing for detecting this new Trojan type along with mutant ranking heuristics to prioritize analysis of the most dangerous functionality. Applying our method to a UART controller design, we discover unspecified and untested bus functionality with the potential to leak 32 bits of information during hundreds of cycles without being detected! Our method also reveals poorly tested interrupt functionality with information leakage potential. After modifying the specification and test bench to remove the discovered vulnerabilities, we close the verification loop by re-analyzing the design using our methodology and observe the functionality is no longer flagged as dangerous.**

I. Introduction

Hardware Trojans are a concern for both semiconductor design houses and the U.S. government [4]. The design, manufacturing, testing, and deployment of silicon chips involves many companies and countries. If a single party involved deems it advantageous to insert malicious functionality into the chip, referred to as *Hardware Trojans*, the consequences can be catastrophic.

Hardware Trojans may be inserted into the 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, leak secret information through side channels, or weaken random number generator output [16].

Many Trojan taxonomies exist [16], [8], [18], 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

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

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

Hardware Trojans modifying unspecified functionality are especially dangerous because even if it were tractable to test a design with all possible input sequences, the correct design behavior for certain cycles is don't care, and any malicious design modifications contained within these cycles are impossible to detect unless the golden behavior is defined.

Category 1 Trojans require triggering circuitry identifying rare design states to prevent the payload from being delivered during testing or normal operation. Identifying design logic activated only under very rare conditions ("nearly unused" logic) is the target of many existing Trojan detection techniques such as FANCI [17], FIGHT [15], VeriTrust [19], and UCI [11].

These techniques are not well suited to detect Category 3 Trojans, because these Trojans can be active without any risk of detection as they do not alter any specified functions. In a modern design, the values of a large percentage of signals during a large number of cycles have no influence on relevant design behavior. Trojan logic hiding in this unspecified space will not match the structure of the "nearly unused" logic required for Category 1 Trojans and targeted by existing detection methods.

Trojans which only affect unspecified functionality are difficult to detect using existing verification methodologies because it is extremely challenging to define and characterize *unspecified* behavior. A hardware specification is often a collection of non-uniform and diverse models such as English documents, state diagrams, input/response test pairs, C, Matlab, or SystemC models, RTL models, cover points, and assertions.

The number of possible malicious modifications to unspecified behavior is limited only by attacker ingenuity. Moreover, an attacker only needs to find a single instance of unspecified functionality not captured by the models listed above to exploit whereas the verification team must anticipate and defend against all possible malicious modifications.

Without an analysis method differentiating dangerous from harmless unspecified functionality, fully specifying design behavior is the only way to ensure a design is not infected with a Category 3 Trojan, however, doing this removes opportunities for optimization and for reasonably complex designs is impossible.

Our proposed analysis methodology addresses these concerns by randomly sampling possible design modifications (known as mutations in mutation testing [12]). 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 Category 3 Trojans.

Our contributions can be summarized as follows: 1) We introduce a new dangerous class of Trojan which leaks information by modifying only unspecified functionality, 2) present a general methodology easily incorporated into existing verification flows to identify **dangerous** unspecified functionality, and 3) apply our method to a UART controller and discover an entire class of Trojan exploiting undefined behavior in bus protocols along with poorly tested interrupt functionality, despite the presence of a sophisticated verification infrastructure.

The rest of the paper is organized as follows: Section II summarizes existing related work on Hardware Trojan detection and mutation analysis, Section III illustrates how a Trojan only modifying unspecified functionality can leak design information, Section IV introduces our methodology for detecting dangerous unspecified functionality, Section V applies our methodology to a UART design, and Section VI summarizes our results and contributions.

II. Related Work

Hardware Trojan Detection: There exist several methods which target the identification of dangerous unspecified functionality at specific abstraction levels. [9] defines unspecified functionality as incompletely specified state transition and output functions, given a digital system specified as a finite-state machine (FSM). The process of logic synthesis takes an incompletely specified FSM M and transforms M to a completely specified gate-level FSM, M', which may contain additional unwanted state transitions and output assignments while still conforming to the original FSM.

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

[9] is limited by the fact that analysis must be performed on a *symbolic* representation of the design state space, the labeling of protected v. non-protected symbolic states must be done manually and it is likely most designers would not have a clear idea or guidance for this labeling task, and either full reachability analysis of protected states is required, making the method unscalable to modern designs, or the T flip-flops in the circuit can be modified so *no* transitions from unprotected to protected states are allowed, which limits design functionality.

[10] introduces a class of Category 3 Trojans which leak information by only modifying RTL don't care bits, and uses combinational equivalence checking techniques to differentiate don't cares which can be exploited by an attacker to leak information and those which are harmless and should remain in the design for optimization during synthesis.

This analysis technique 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.

We build upon the ideas presented in [10], but our proposed method is more general than both [9] or [10] since we base our technique on mutation testing, which 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.

Mutation Testing: 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 [12], [7].

In the hardware domain, mutation testing is primarily used for test bench qualification [6]. Fault models and fault injection tools exist for SystemC [13], TLM [5], and RTL [1].

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 [14] 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.

III. Information Leakage Trojans

Consider the simple FIFO module in Figure 1. Data is written to the FIFO when it is not full and `write_enable == 1`, and data is read from the FIFO when the FIFO is not empty and `read_enable == 1`. Data can be written to and read from the FIFO simultaneously.

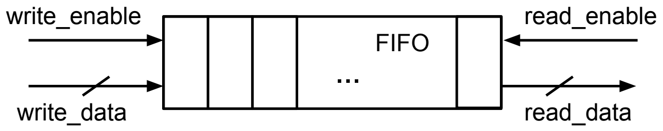

Fig. 1: Simple FIFO

What is the correct value of `read_data` when `read_enable == 0`? To save energy, it would make sense to maintain the value of `read_data` from the last read operation, but it is unlikely that this particular behavior is explicitly specified or tested. Figure 2 shows a simple Trojan which leaks information by setting `read_data` to other internal design signals whenever a read operation is not occurring.

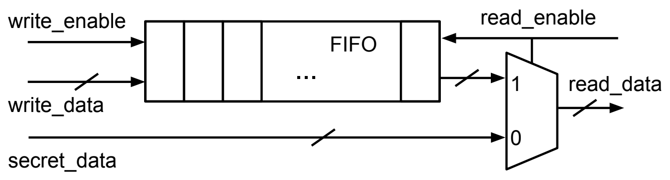

Fig. 2: FIFO with Trojan

To illustrate the potential of mutation analysis to detect this Trojan, consider a fault which changes the AND operator (highlighted in gray) to an OR operator in Line 2 of Listing 1. 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 1 and in the following cycle, even if `read_enable == 0`, `read_data` will be updated with the value of the next FIFO item.

Listing 1: 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 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 (`read_data`) 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) and 2) the fault is undetected.

The detection methodology we present in the following section 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 Figure 2.

IV. Detection Methodology

A. Threat Model and Mutant Selection

Our method assumes the attacker can modify the design at the RT abstraction level or higher. This is because Trojans inserted in the gate-level netlist or later design stages (meaning there is a golden RTL model available) can be detected using commercial equivalence checking tools which exist as part of the standard chip design flow once the X-analysis method proposed in [10] is applied to ensure any X's in the golden RTL model cannot be used to implement Category 3 Trojans.

As in [10], we assume the goal of the Category 3 Trojans our method targets is to leak information to signals at the boundary of the IP core being analyzed or to internal registers/signals the attacker can observe. Since we identify scenarios where the test bench is unable to detect changes in attacker-observable or IP boundary signals, any Category 3 Trojans performing other functions affecting these signals besides information leakage are also targeted by our method.

Our methodology can be performed at any level of abstraction, but since identifying unspecified design behavior is central to preventing Category 3 Trojans, performing analysis at abstraction levels describing the design behaviorally as oppose to structurally makes interpreting the results of mutation testing easier. Moreover, simulation speed improves when more detail is abstracted away from the design, and any improvements in the design specification or functional tests resulting from our technique carry over to all subsequent refined versions of the design relying on the same verification infrastructure.

To clarify, our technique does not directly mark specific lines of code, wires, or gates as being part of a Trojan, in the same way that mutation analysis does not directly find bugs. Rather, our analysis technique highlights design functionality that is susceptible to Trojan insertion and calls for refinement of the specification or more thorough testing of the at risk functionality. If a Trojan already exists in this functionality, then it is likely that the test bench improvements will detect the Trojan, and if there is no Trojan, the improvement effort increases the chances that Trojans inserted in this functionality later on in the design life cycle will be detected.

Given the wide variety of mutation models available [12] what is the criteria for selecting the best model for detection of Category 3 Trojans, and can mutation analysis aid in detection of other Trojan types?

An underlying assumption of mutation analysis is the Coupling Effect Hypothesis [12]: "Complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults." The example in Section III illustrates this concept because the more complex fault (the Trojan in Figure 2) can be detected if the simpler fault (highlighted in Listing 1) is detected.

If our method targeted Trojans with rare triggering conditions (Category 1 Trojans), this hypothesis would not hold, since these Trojans are examples of pathological faults. However, since information leakage Trojans are most effective when they affect a large number of observable signals during a large number of cycles, the uniform structural sampling that simple mutation models provide (such as those used by Certitude [1]) should be effective enough to highlight the most vulnerable unspecified functionality if enough faults are injected. A more thorough analysis of the effect different mutation models have on the success of our technique is a topic for future research.

B. Mutant Injection and Analysis

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 3 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 Figure 3.

Identifying Attacker-Observable Signals: 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. 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 Figure 3)? 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! Coverage Discounting [14] 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 Figure 3 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.

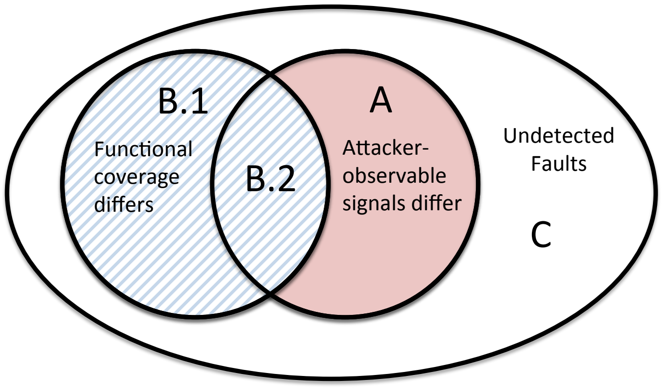
B.1
Functional coverage differs
B.2
A
Attacker-observable signals differ
Undetected Faults
C

Fig. 3: Scenarios for Undetected Faults

C. Ranking Undetected Faults

Although less than the total number of undetected faults, the number of undetected faults in Region A of Figure 3 can still be too costly to completely analyze.

It is desirable that the undetected faults highlighting unspecified functionality most advantageous for an attacker to exploit be analyzed first. Since mutation analysis is an iterative process, after improving the test bench so that it is capable of detecting the most dangerous faults, it is likely that many other previously undetected faults will now be detected and not require analysis.

If there are not enough resources to dedicate to ensuring that all undetected faults are eventually detected, the proposed ranking metrics provide confidence that precious man-hours are spent analyzing only the most severe threats.

The following metrics are easily observed during the injection of each fault:

1) Number of attacker-observable bits differing
2) Total time attacker-observable signals differ
3) Number of *distinct* tests producing differences in attacker-observable signals

Metric 1 captures whether the fault affects a few specific signals or broadly impacts the set of attacker-observable signals. For example, if a design has 10 attacker-observable signals, and Fault A causes 8 to differ at some point during testing, while Fault B only causes 3 to differ, a Trojan related to the functionality of Fault A can potentially leak more bits of information during a given cycle.

However, the number of cycles a given signal differs is also proportional to the information leakage potential of a Trojan based on a particular fault. Metric 2 is the sum over all tests and all attacker-observable signals of the total time each signal differs. If the 8 signals differing under Fault A only differ for 2 cycles each, while the 3 signals differing under Fault B differ for 10 cycles during testing, a Trojan formed from Fault B may be more useful to an attacker.

Metric 3 gauges how likely information leakage will occur under normal usage scenarios. Presumably, the verification tests at the very least exercise typical design functionality. If the attacker cannot force the design into states activating the mutated functionality, faults that lead to observable differences across many tests are more useful for developing Trojans which provide information leakage capabilities during more design states.

D. Method Overhead and Coverage

One well known disadvantage of mutation testing is the long runtime required to apply the entire test suite to each faulty version of the design. While our methodology also requires simulation of all tests for each mutant, the additional overhead needed to record attacker-observable signals and coverage, then compute differences with the fault-free design, is negligible in comparison.

Another disadvantage of mutation testing is the amount of manual effort required to analyze each undetected fault. The fault ranking mechanism presented in Section IV-C somewhat alleviates this problem by allowing the verification engineer to review the functionality with the largest risk of hiding an information leakage Trojan to be analyzed first. Once the most dangerous fault is detectable by the test bench, all other dangerous faults can be re-evaluated, and those which are now detected due to test bench improvements no longer need to be analyzed.

The simplest and most effective method of decreasing the runtime of mutation analysis is to simulate fewer faults. A method identifying the minimal fault set required to expose all verification holes for a design unfortunately does not exist. Developing metrics to determine when a sufficient number of mutants have been simulated is one of the hardest problems to address in mutation testing. Future research will investigate how the metrics in Section IV-C and additional simulation data can be used to develop these necessary metrics.

V. UART Controller Case Study

We analyze a UART (universal asynchronous receiver/-transmitter) design from OpenCores [2] using the methodology presented in Section IV. After analysis of just 4 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. We now provide the case study details.

The test bench used in this case study is a propriety OVM-based suite provided by an EDA tool vendor consisting of

110 Undetected Faults		
Region A	**Region B**	**Region C**
30 faults	2 faults	78 faults

TABLE I: Categorization of Undetected Faults

80 directed tests with contained 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 [1]. 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.

Fault Classification: 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 Figure 3 is presented in Table I. Using our methodology, the number of faults requiring manual analysis (those in Regions A and B) is reduced from 110 to 32!

A. Wishbone Bus Trojan

The 3 ranking metrics presented in Section IV-C are equally weighted to identify the most dangerous faults. The 3 most dangerous faults (1411, 1412, and 1413) all affect the following line, which assigns the output enable control bit to 1 in the Wishbone Bus [3] interface if all 4 conditions are true:

Listing 2: Assignment of Output Enable

```
assign oe = ~wb_we_is & wb_stb_i & wb_cyc_i & wbstate
    ==2'b01;
```

Each of the 3 faults changes 1 of the bitwise AND operators (highlighted in gray) to a bitwise OR operator. For example, fault 1411 changes the assignment to:

Listing 3: Fault 1411

```
assign oe = ~wb_we_is & wb_stb_i & wb_cyc_i | wbstate==2'
    b01;
```

effectively setting `oe` whenever `wbstate==2'b01`, even if another condition is false, which in the original fault-free design, would have prevented `oe` from being set.

When `oe` is set, the 8-bit data bus lines (coming from the UART register file) are re-sampled, and placed in the correct byte lane on the 32-bit data output bus (`wb_dat_o`) as seen in the following code:

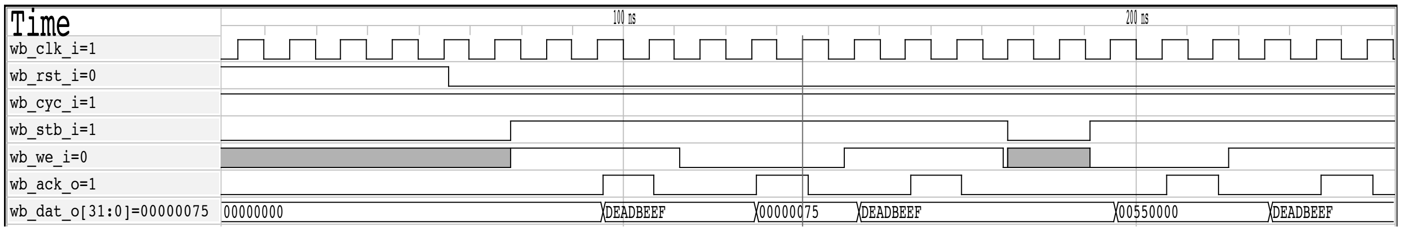

Fig. 4: Output Enable Trojan Waveform for Bus Protocol Test

Listing 4: Assignment of Data Output Bus

```
if (oe)
  case (wb_sel_is)
    4'b0001:wb_dat_o <= {24'b0,wb_dat8_o};
    4'b0010:wb_dat_o <= {16'b0,wb_dat8_o,8'b0};
    ...
```

If `oe` is not set, the data bus retains its previous value.

Why Are Faults Modifying `oe` Undetected? To understand why spurious changes on the data output bus are not detected by the test bench, which includes a Wishbone bus protocol checker, we must elaborate on the functionality of the protocol control signals involved in the assignment of `oe` (Listing 2): `wb_stb_i` (STB_I in the bus specification document), `wb_we_is` (WE_I), and `wb_cyc_i` (CYC_I).

From the specification, STB_I, set by the bus master, selects a particular slave, and "a SLAVE shall respond to other WISHBONE signals only when this [STB_I] is asserted...", "the cycle input [CYC_I], when asserted, indicates that a valid bus cycle is in progress", and "the write enable input [WE_I] indicates whether the current local bus cycle is a READ or WRITE cycle" [3].

In the original design, `oe` is only set during a valid read transaction. Under the 3 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. However, 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.

Example Output Enable Bus Trojan: 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 all the conditions in Listing 2 are not simultaneously met and the test bench would be none the wiser! Specifically, the Trojan can take advantage of the fact that the value of `wb_dat_o` is **unspecified** during a write transaction or invalid cycle.

We implement this bus Trojan by changing the assignment of `oe` to match Listing 3. We then choose to leak the value `0xdeadbeef` over the bus only during write transactions and invalid cycles by modifying the code in Listing 4 to the following (the Trojan is Lines 2-3):

Listing 5: Output Data Bus Trojan

```
1  if (oe) begin
2    if (wb_we_i | ~wb_stb_i | ~wb_cyc_i)
3      wb_dat_o <= 32'hdeadbeef;
4    else
5    case (wb_sel_is)
6      4'b0001:wb_dat_o <= {24'b0,wb_dat8_o};
7      4'b0010:wb_dat_o <= {16'b0,wb_dat8_o,8'b0};
8      ...
```

Figure 4 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 135ns, 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 Line 3 in Listing 5 can be changed to assign *any* 32-bit value 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 by Figure 4. It is very unlikely that this Trojan would be detected by existing methods targeting the identification of rarely used logic.

Improving the Bus Checker: To detect faults 1411, 1412, and 1413, the following additional check is added to the existing bus protocol checker: `wb_dat_o` can not change unless the design has been reset, or a read request is being acknowledged. In addition to detecting the 3 faults, the Output Enable Bus Trojan is also detected.

In a traditional verification setting, it would be unnecessary and cumbersome to add this additional check, and the 3 faults would be considered a waste of time to analyze because they do not affect the correctness of normal read/write operations. Our work is the first 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 in Section III actually belongs to this class of Trojan.

B. Interrupt Output Signal

After improving the test bench to detect the 3 faults related to the Wishbone bus, the ranking metrics presented in Section IV-C identify Fault 918, which affects the interrupt mechanism, as the most dangerous fault. This fault affects *specified* functionality, and belongs to Region B.2 in Figure 3. 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 5 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.

Fault 918 causes `int_o` to become unknown for many cycles during 49 of the 80 tests. More specifically, Fault 918 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 10 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 Fault 918 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.

VI. Conclusion

We illustrate the danger of Trojans which leak information by only modifying unspecified design functionality, and show why current verification and Trojan detection methodologies are unable to detect this class of Trojan. We propose an automated detection methodology for this Trojan type based on mutation testing. Our method is applicable to a wide range of abstraction levels, and also works to identify poorly tested specified functionality in addition to dangerous unspecified functionality. We demonstrate the effectiveness of our approach by finding an entire class of information leakage Trojans related to unspecified bus functionality after analyzing the 3 most dangerous faults highlighted by our method in a UART controller design. Our method also led to the discovery of poorly tested interrupt functionality vulnerable to Trojan insertion. We then close the verification loop by improving the design checkers to detect these faults and show that this improvement leads to the detection of an actual Wishbone bus Trojan.

References

[1] Synopsys Certitude: https://www.synopsys.com/TOOLS/VERIFICATION/FUNCTIONALVERIFICATION/pages/certitude-ds.aspx.

[2] UART 16550 Core. http://opencores.org/project,uart16550.

[3] Wishbone Bus. http://opencores.org/opencores,wishbone.

[4] S. Adee. The Hunt for the Kill Switch. *IEEE Spectr.*, 45(5):34–39, May 2008.

[5] N. Bombieri et al. A Mutation Model for the SystemC TLM 2.0 Communication Interfaces. In *DATE*, 2008.

[6] N. Bombieri et al. Functional Qualification of TLM Verification. In *DATE*, 2009.

[7] B. Breech, M. Tegtmeyer, and L. Pollock. An Attack Simulator for Systematically Testing Program-based Security Mechanisms. In *ISSRE*, 2006.

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

[9] C. Dunbar and G. Qu. Designing Trusted Embedded Systems from Finite State Machines. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(5s):153, 2014.

[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] M. Hicks et al. Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically. In *IEEESSP*, 2010.

[12] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[13] P. Lisherness and K.-T. Cheng. SCEMIT: A SystemC Error and Mutation Injection Tool. In *DAC*, 2010.

[14] P. Lisherness, N. Lesperance, and K.-T. Cheng. Mutation Analysis with Coverage Discounting. In *DATE*, 2013.

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

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

[17] A. Waksman, M. Suozzo, and S. Sethumadhavan. FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis. In *SIGSAC*, 2013.

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

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