Mining Mutation Testing Simulation Traces for Security and Testbench Debugging

Nicole Fern
University of California Santa Barbara
Email: nicole@ece.ucsb.edu

Kwang-Ting (Tim) Cheng
Hong Kong University of Science and Technology
Email: timcheng@ust.hk

***Abstract*—Unspecified design functionality can be modified by Hardware Trojans to leak information. Existing methods capable of detecting these Trojans require that unspecified functionality already be characterized, and suggest a manual ad-hoc process to enumerate "don't care" conditions potentially containing security vulnerabilities. Prior work has shown the potential of mutation testing to uncover testbench holes and highlight unspecified functionality, but requires tedious manual analysis of undetected faults to gain useful insight. This work provides the missing link required to fully automate characterization of unspecified functionality and can formally prove the absence of Trojans. Our approach is to mine simulation traces generated during mutation testing to produce assertions characterizing verification holes or unspecified functionality. These assertions can be fed directly to Trojan detection methods making securing unspecified functionality a completely automated process. Our trace mining technique is able to identify unspecified Wishbone bus functionality in a Trojan-free UART core and verify the functionality is benign, while flagging the same functionality in a Trojan-infected version of the design.**

I. Introduction

Verification is a bottleneck in the hardware design process, estimated to consume over 70% of hardware development resources. Developing pre-silicon verification infrastructure is crucial in order to catch functional bugs before tape-out and avoid silicon re-spin, however the process requires immense manual effort and is largely ad-hoc. Moreover, traditional verification techniques focus on increasing confidence that specified functionality is correct. Unmodeled behavior will not be verified by existing methods, meaning any *security vulnerabilities* in unspecified functionality will go unnoticed.

Security vulnerabilities include accidental bugs as well as malicious functionality (Hardware Trojans) inserted by an attacker with access to the design. Chip design mandates thousands of engineers and an entire ecosystem of design tools and fabrication services have access to the design, making Trojans a major concern for both the semiconductor industry and governments recognizing the dependence of critical infrastructure on off-the-shelf electronics [4], [20], [13].

In this work we focus on detecting Trojans inserted in the design *pre-silicon* (meaning no "golden" design model exists) whose behavior is embedded completely within *unspecified functionality*. For example, Trojans which use the existing on-chip bus infrastructure to leak information during idle bus cycles [6] never interfere with normal bus transactions and are hard to detect with functional tests.

The number of cycles during which signals are "don't care" in modern designs is large due to increasing design complexity. At the Register Transfer Level (RTL) or above, it is often impossible to enumerate what the desired value of every signal should be at every cycle and even more impractical to expend verification effort on "don't care" functionality. This provides ample opportunity for Trojans to implement malicious behavior even when constrained to only modifying unspecified functionality.

Prior work addressing the same threat model provides methods to detect such malicious circuitry [7], but *only after unspecified functionality is characterized*. Characterization is tricky because by definition unspecified functionality encompasses aspects of the design outside the focus of project engineers. Additionally, categorizing design states as specified or unspecified requires knowledge of the verification infrastructure such as the test stimulus chosen to exercise the design, functional coverage, checkers, and assertions, which all function to explicitly highlight important design behavior.

The approach taken by [5] to address these challenges is to use mutation testing [10], whose primary purpose is providing a metric to gauge testbench quality, to highlight unspecified functionality vulnerable to Trojan insertion by identifying blind-spots in the verification infrastructure. These blind spots are either verification holes or correspond to unspecified functionality, but in both cases require analysis by a verification engineer. The amount of manual effort required to analyze every fault not detected by the verification infrastructure is a known drawback of mutation testing, and the technique proposed in [5] suffers from this limitation as well.

Our main contribution is providing the missing piece: automated interpretation of mutation testing results in the form of assertions, which can be used as input to existing formal Trojan detection techniques based on solving satisfiability problems such as [7] without requiring any human intervention. Figure 1 highlights in gray the aspect of the Trojan detection workflow our technique automates.

The detection technique proposed in [7] only analyzes the design under conditions when input or internal signals are unspecified. Another contribution of this work is providing a formal method capable of identifying if information is being leaked from a design during conditions when output signals are unspecified (output "don't cares").

Our approach is to mine assertions describing conditions

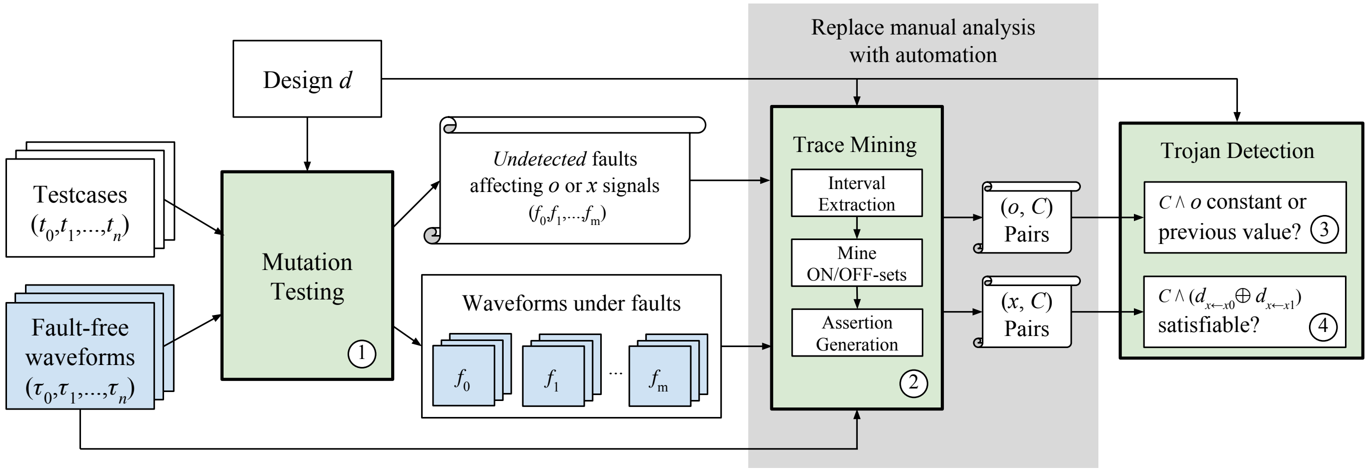
Replace manual analysis with automation
Design d
Testcases (t_0,t_1,...,t_n)
Fault-free waveforms (τ_0,τ_1,...,τ_n)
Mutation Testing
①
Undetected faults affecting o or x signals (f_0,f_1,...,f_m)
Waveforms under faults
f_0
f_1
...
f_m
Trace Mining
Interval Extraction
Mine ON/OFF-sets
Assertion Generation
②
(o, C) Pairs
(x, C) Pairs
Trojan Detection
$C \wedge o$ constant or previous value? ③
$C \wedge (d_{x\leftarrow x0} \oplus d_{x\leftarrow x1})$ satisfiable? ④

Fig. 1. Detecting Trojans in Unspecified Functionality by Mining Mutation Testing Results

under which a target signal, either an output (o) or input/internal (x) signal, *differed* under *undetected* faults from the waveforms produced during fault simulation. These assertions describe when a signal is either unspecified or poorly verified, because the signal differed in the faulty and fault-free design versions under the test stimulus, but the verification infrastructure was unable to detect this difference.

Each assertion, $\mathcal{C}$, and corresponding (o, $\mathcal{C}$) or (x, $\mathcal{C}$) pair is classified as dangerous or safe by automated formal Trojan detection techniques. Only dangerous pairs need to be further scrutinized as safe pairs are formally proven to be benign. The counterexamples returned by the formal analysis tools along with the dangerous assertions provide additional information to guide the engineer in debugging the testbench or locating the Trojan. To summarize, our contributions are:

- Condensing simulation trace information from multiple undetected faults and tests into assertions for efficient testbench debugging (② in Figure 1).
- Automated characterization of unspecified functionality allowing prior detection techniques [7] (④ in Figure 1) to be employed in a completely automated manner.
- Providing a method for detecting Trojans leaking information through output don't cares (③ in Figure 1).

The rest of the paper is organized as follows: Section II reviews related work in assertion mining and Trojan detection, Section III details our threat model, Section IV describes the trace mining methodology in detail, Section V details how the mined assertions are used to detect Trojans, Section VI applies our methodology to a UART controller, and in Section VII we summarize our results and contributions.

II. Related Work

Assertion Mining: Techniques such as Goldmine [18], [12] and Scalable Assertion Miner (SAM) [11] extract design specifications from simulation traces in the form of assertions. We adopt a strategy similar to Goldmine, and use signal valuations seen in the simulation trace to construct a decision tree from which assertions are created, however the assertions mined by our technique are fundamentally different than those mined by Goldmine and SAM. Goldmine and SAM produce assertions reflecting design behavior, while our technique characterizes conditions under which signals differ between the original design and versions of the design where artificial faults have been injected. Essentially, we are using assertion mining to characterize the differences between two design versions instead of characterizing the design itself. In [11], a fault localization scheme is also proposed, however our goal is not to localize the error (we already know exactly which line of code the fault is injected in because it is artificial), but determine *when* the error affects signals in the design, requiring a completely different approach.

Hardware Trojan Detection: Existing detection techniques for Trojans inserted pre-silicon often target violations of *specified* behavior occurring under extremely rare conditions, where the main challenge is identifying these conditions [19], [16], [21], [9]. These techniques are not well suited to detect Trojans in unspecified functionality as Trojans which never violate design specifications have no need to hide in almost unused logic. Prior work specifically addressing Trojans in unspecified functionality can detect the presence of Trojans in manually characterized unspecified functionality [7] and identify unspecified functionality using mutation testing [5], however, [7] is unable to analyze outputs under unspecified conditions, and [5] requires manual analysis of faults to both detect Trojans and address benign unspecified functionality. Our work overcomes these shortcomings by providing a method to analyze the design under output don't care conditions and automating the process of identifying unspecified functionality and verifying it is Trojan-free.

III. Threat Model

This work addresses Trojans which only modify unspecified functionality. We assume that Trojans can be inserted in the design RTL itself, and do not require a golden model. We also

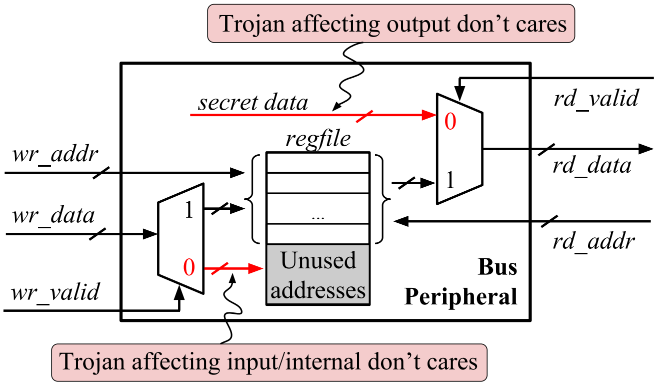

Fig. 2. Trojans in Unspecified Bus Functionality

assume the Trojan operates in the digital domain, meaning Trojans which leak information through side-channels (ex. power consumption) are not addressed in this work.

Unspecified functionality for a module or IP core can be divided into two categories:

1) **Output Don't Cares:** Conditions during which the value of module output signals do not affect design functionality. Examples include idle bus cycles, intermediate cycles of an arithmetic computation, and cycles where the module is not enabled or clock-gated.
2) **Input/Internal Don't Cares:** Conditions during which the values of input or internal signals are assumed to never affect the design state or outputs during output care conditions. Examples are unused register bits, bus inputs during idle cycles, and memory/fifo data signals when reads/writes are not occurring.

Trojans can leak sensitive information to circuit boundaries or software-visible registers and memory locations by modifying signals under don't care conditions. For example, [6] illustrates how memory accesses made by the root user in a Linux system can be made observable to an unprivileged program by inserting a Trojan which modifies on-chip bus interface signals only during cycles when the bus is idle. Figure 2 provides a simplified example of a Trojan-infected design conceptually similar to the Trojans in [6]. The peripheral in Figure 2 is a bus slave, meaning its inputs are controlled by a bus master capable of initiating read and write transactions to the peripheral's register file. If the read and write valid signals are 0, no transactions occur and the bus is idle. The values of the data and address lines during idle cycles are *unspecified*.

The condition $wr\_valid = 0$ is an input don't care condition for *wr_data* (write data) and *wr_addr* (write address). Normally, when *wr_valid* is 0, no data is written to the register file, however the Trojan circuitry (shown in red in Figure 2) uses this condition to store *wr_data* to an unused location in the register space. Unused bits within existing registers can also be used to store data covertly. A Trojan in the bus interconnect fabric spying on another component's transactions can route the leaked data to *wr_data* only during idle cycles with the end goal being data retrieval via a malicious software program with access to the peripheral.

The condition $rd\_valid = 0$ is an output don't care condition for *rd_data* and Trojan circuitry (shown in red in Figure 2) leaks an internal design signal, *secret data*, to *rd_data* only under this don't care condition. The Trojan takes advantage of the fact that when *rd_valid* is 0 no bus master is currently reading from the peripheral. Bus masters and verification testbench checkers both don't care about the value of *rd_data* during idle cycles, and this allows an attacker to retrieve information discreetly.

In order to guarantee detection of all possible Trojans modifying don't cares, for every input, output, and internal signal, the conditions under which that signal is unspecified must be completely characterized. This is a challenging requirement, and for most designs impossible to satisfy. Our mining methodology produces an assertion set capturing when a signal is unspecified utilizing the existing verification infrastructure. The assertion set will become more complete as the testbench itself is improved and as more faults are injected. This provides a way to improve confidence that the design is Trojan-free given a finite amount of simulation cycles and man-hours to dedicate to the verification and Trojan detection task.

IV. Mining Methodology

Figure 1 shows the information gathered from mutation testing used as input to the trace mining process. Mutation testing tools, such as [1], inject faults by making syntactic changes in the design source code (for example changing an & to an | operator or removing the right-hand side of an assignment), then for each faulty version of the design run test cases provided by the testbench. Waveforms containing the values of signals in the design at different points in time are generated during each test, meaning the fault-free and all faulty versions of the design can be compared.

For each testcase, the verification infrastructure has a mechanism for determining if the test passes or fails. Before mutation testing it is assumed that all tests pass for the fault-free version of the design. If any test fails while simulating a fault, the fault is *detected*. If *all* tests pass then the fault is *undetected*, meaning the fault-free and faulty designs are identical from the viewpoint of the testbench. From the set of *undetected* faults, our mining methodology only examines those for which signals of interest (o or x signals) differed during fault simulation.

Fault Selection: Selecting the optimal set of faults to maximize discovery of actual design bugs is a non-trivial task and an unsolved problem in mutation testing [10]. For detecting Trojans in unspecified functionality, the fault set should explore the maximum design space in the most uniform manner because both the location of the Trojan and unspecified functionality are unknown beforehand. In this work, the fault set is formed from a collection of simple syntactic changes to each line of code in the design. This provides uniform design coverage, and the number of faults per line of code

TABLE I
EXAMPLE SIMULATION TRACES τ_i (FAULT-FREE) AND $\tau_{i,j}$ (FAULTY) FOR SIGNALS a, b, c, AND x

		Time (in cycles)									
Signal		1	2	3	4	5	6	7	8	9	10
x	$\tau_{i,j}$	0	1	0	0	1	0	1	1	1	1
	τ_i	0	0	1	0	0	1	0	0	1	1
a	τ_i	0	1	0	0	1	1	0	1	0	0
b	τ_i	0	1	1	0	0	0	1	1	0	0
c	τ_i	0	1	0	1	1	0	1	0	1	0

is an adjustable parameter used to trade-off exploration of the design space with simulation time.

Selection of o and x signals: The goal of our mining methodology is to characterize unspecified functionality, which is comprised of output don't cares and input/internal don't cares (definitions of these categories are given in Section III). o signals are design outputs, and all should be analyzed, making the remaining signals x signals. It is impractical to analyze all possible x signals, so some ideal candidates are software-visible design registers and control-path/configuration state elements.

Undetected faults affect aspects of the design not monitored by the testbench, hence if we are able to mine the conditions under which a target signal differs under a set of faults we have identified when the target signal is unspecified and can verify the absence of malicious behavior during the mined conditions. The undetected faults affecting target signals can be grouped into sets for each target signal. On one end of the grouping spectrum, all faults affecting the target signal form a single set, and on the other end each fault is in a set of its own. A motivation for grouping faults is that mining accuracy improves with more simulation data, however if the fan-in of a target signal encompasses a large portion of design functionality (ex. a bus data signal), analyzing faults affecting different aspects of the design in a single fault set may lead to overly complex assertions.

The following subsections correspond to the blocks in ② in Figure 1 and detail the trace mining procedure run for each target signal and fault set affecting the signal. The output of the mining process is a list of $(o, \mathcal{C})$ and $(x, \mathcal{C})$ pairs, which can be processed by the Trojan detection methodology detailed in Section V.

A. Difference Interval Extraction

Given a target signal and fault set, the first step in the mining process is to extract a list of time intervals for each test where the target signal differed from the fault-free version under a fault in the fault set. This is accomplished by extracting difference intervals for each (fault, test) pair then combining the intervals for all faults in the fault set.

The top row in Table I corresponds to a trace for test i for signal x recorded during the simulation of fault j. The second row is the fault-free version of the trace. If x is the target signal, the difference intervals extracted for test i are $2-3$, and $5-8$.

B. ON and OFF-set Mining

Once difference intervals have been identified for each test, **fault-free** traces for each test are used to build a function, F. We define the *predictor set* (p) for the target signal as the domain of F. Signals in the fan-in cone for the target signal form the predictor set. If the fan-in cone is large, the number of predictors can be limited to fan-in signals closest to the target signal. The goal is to mine function F so that the ON-set (set of values for variables in p where $F(p) = 1$) characterizes when the target signal is unspecified.

When analyzing the fault-free waveform for each test, each time step either lies within a difference interval or not. If the time step lies within a difference interval, the values of the predictor signals during that time-step, from now on called a row, are added to the ON-set for F, otherwise the row is added to the OFF-set. Only fault-free waveforms are used to mine F because characterization of unspecified functionality should be expressed in terms of the original design as this is the design that will eventually be fabricated, not the faulty version. Of interest are conditions seen in the original design under which the target signal value can change without the testbench noticing.

As an example, the fault-free trace, τ_i, given in Table I, is mined to obtain F using predictors a, b, and c. The columns in Table I corresponding to ON-set rows are highlighted in green and columns corresponding to OFF-set rows in blue. The resulting ON-set is $\{111, 010, 101, 100, 011, 110\}$ and the OFF-set is $\{000, 001\}$.

The ON-set and OFF-set must be distinct, meaning a row cannot appear in both the ON-set and OFF-set. A major difference between our mining technique and traditional assertion mining is that in the latter, conflicts can never occur because behavior of a signal in a single design instance is mined. In our application every fault corresponds to a different design version. Conditions causing the target signal to differ under one fault may not cause the signal to differ under another, but this doesn't mean the condition corresponds to specified behavior, just that some fault(s) did not affect the target signal under the condition. Therefore, if a row appears in both the ON-set and OFF-set, it is kept in the ON-set and deleted from the OFF-set.

The accuracy and completeness of the mined function F depends on the percentage of function behavior explored by the test stimulus, and the impact of the fault set on the target signal. For the example in Table I, there are only 3 predictors making it likely all 8 possible rows will appear in the traces. When more predictors are used, it can be assumed some rows are never observed. This is one source of inaccuracy in the mining process, and a well known limitation of assertion mining techniques. Another source of inaccuracy unique to mining difference intervals is that even if all possible rows are observed, a row may correspond to unspecified functionality not activated by any faults in the fault set, meaning it is added to the OFF-set, but under a more complete fault set would be seen within a difference interval and belong in the ON-set.

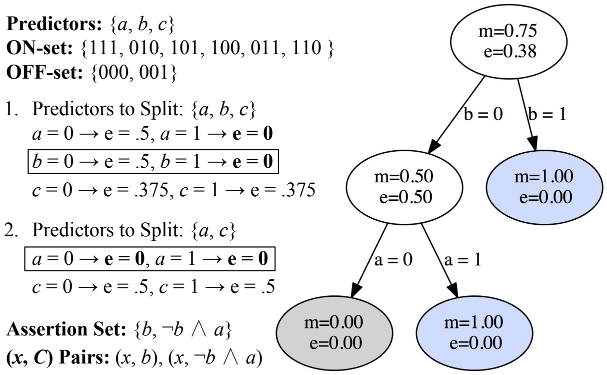
Predictors: {a, b, c}
ON-set: {111, 010, 101, 100, 011, 110 }
OFF-set: {000, 001}
1. Predictors to Split: {a, b, c}
a = 0 → e = .5, a = 1 → e = 0
b = 0 → e = .5, b = 1 → e = 0
c = 0 → e = .375, c = 1 → e = .375
2. Predictors to Split: {a, c}
a = 0 → e = 0, a = 1 → e = 0
c = 0 → e = .5, c = 1 → e = .5
Assertion Set: {b, ¬b ∧ a}
(x, C) Pairs: (x, b), (x, ¬b ∧ a)
m=0.75
e=0.38
b = 0
b = 1
m=0.50
e=0.50
m=1.00
e=0.00
a = 0
a = 1
m=0.00
e=0.00
m=1.00
e=0.00

Fig. 3. Example Assertion Generation and Decision Tree Construction

C. Assertion Generation

While every row in the ON-set can theoretically produce an assertion for the target signal, there is often opportunity to reduce redundancy in the assertion set. For example, the ON-set for $F(a, b, c)$ mined from Table I, $\{$101, 110, 100, 111, 010, 011$\}$, can be completely described by two assertions: a, and $(\neg a \wedge b)$ or alternatively b, and $(\neg b \wedge a)$. The variable c is not necessary to determine when x is unspecified. Reducing the number of assertions produced by our mining methodology is essential as each assertion is analyzed by formal methods. Additionally, if the detection techniques classify the assertion as dangerous, the assertion must manually be examined. Fewer assertions results in reduced formal method runtime and manual analysis effort.

A compact set of assertions is produced by building a *decision tree* classifier using the ON and OFF-sets as training data. Decision trees [15], [14] are used to extract underlying rules and patterns and organize data. For assertion mining, the ON and OFF-set rows of F are compressed into a tree structure where each node corresponds to a predictor, outgoing edges represent the value (either 0 or 1) of the predictor, and leaf nodes indicate membership in either the ON-set or OFF-set. The values of predictors along the path from the tree root to a leaf node form a series of constraints which must be satisfied for any row to be in the class indicated by the leaf node.

The most important factor in decision tree construction is the order in which predictors are selected to split the data. This is done in a "greedy" fashion by finding the predictor, p, resulting in the smallest average error amongst partitions of the data divided according to possible predictor values (in our case either 0 or 1). The first step is to compute the mean class value for each partition, $m = |\text{ON-set}| \div (|\text{ON-set}| + |\text{OFF-set}|)$, where $|\quad|$ is the number of rows in the ON/OFF-set. The error for each partition, e, and average error for all the partitions, e_{avg} is given by:

$$e = \frac{m|\text{OFF-set}| + (1-m)|\text{ON-set}|}{|\text{ON-set}| + |\text{OFF-set}|}, \;\; e_{avg} = \frac{e_{p=0} + e_{p=1}}{2}$$

The predictor with the smallest average error is selected to create outgoing edges from the current node being processed. If a partition results in zero error, the edge points to a leaf node, where the average class value is either 0 (OFF-set) or 1 (ON-set). The tree construction algorithm prioritizes splitting on the predictors most relevant in distinguishing when the target signal is specified versus unspecified and prunes irrelevant predictors. Tree construction terminates when all paths from the root end in a leaf node (zero error) or there are no more predictors to split on. The resulting tree is completely consistent with the original ON-set and OFF-set data.

Figure 3 provides an example of how a decision tree is built from the ON-set and OFF-set for F. The root node is labeled with the overall mean and error of the data. Since a and b both have an average error of $.25$, b is arbitrary chosen as the predictor to split the data on. When $b = 1$, all rows belong to the ON-set and the $b = 1$ edge points to a leaf node. When $b = 0$, the error is non-zero, and another predictor must be constrained to determine the class. a is selected, and since both $a = 0$ and $a = 1$ partitions result in zero error, the tree building algorithm terminates.

To generate assertions, the tree is traversed and each path leading to a leaf node corresponding to the ON-set becomes a condition (conjunction of literals) describing when the target signal is unspecified. Ideally, the number of assertions generated from the decision tree is significantly less than the number of rows in the ON-set, and there is a reduction in the number of predictors.

Assertion Ranking by Tree Depth: Another advantage of assertion set compression using decision trees is that leaf node depth can be used as a metric to prioritize analysis of assertions. Assertions generated from short paths to leaf nodes contain a smaller number of literals relative to the total number of predictors present in the tree therefore describe a larger percentage of unspecified functionality compared with complicated assertions corresponding to more constrained situations where the target signal is unspecified. Additionally, assertions generated from shorter paths are less likely artifacts of incomplete simulation data making these assertions a higher priority for analysis.

V. Trojan Detection

Trace mining generates a list of assertions describing conditions under which a target signal is unspecified. The goal of Trojan detection is to formally analyze design behavior under these conditions to verify no malicious actions involving the target signal occur.

A. Input/Internal Don't Cares

If an input/internal signal, x, unspecified under condition $\mathcal{C}$, propagates to design outputs under $\mathcal{C}$, the signal can maliciously leak information. The technique proposed in [7] (④ in Figure 1) detects if x can influence outputs in design d under $\mathcal{C}$ by determining the satisfiability of Equation 1.

$$\mathcal{C} \wedge (d_{x \to x_0} \oplus d_{x \to x_1}) \qquad (1)$$

If satisfiable, there exists a pair of different values for x (x_0 and x_1) which cause differences in the output under $\mathcal{C}$

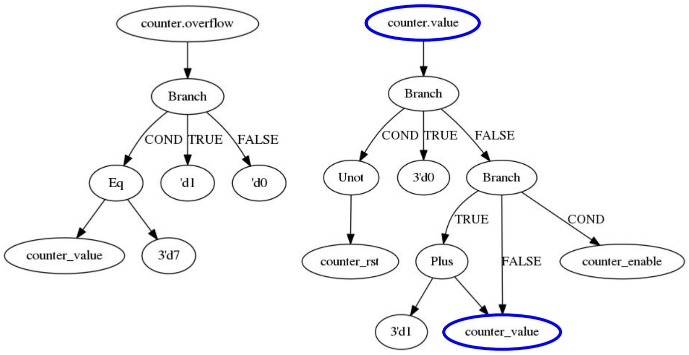

Fig. 4. Data-flow Graphs for Overflow Signal (left) and Counter Value Signal (right)

meaning x is likely involved in Trojan circuitry or a serious design bug exists.

The Trojan covertly writing to unused register space in the example bus peripheral shown in Figure 2 would be detected by determining that Equation 1 is satisfiable when analyzing the (x, $\mathcal{C}$) pair (x = *wr_data*, $\mathcal{C} = \neg$*wr_valid*). The counterexample provided by the satisfiability solver would concretely show that *wr_data* can propagate to the output of the bus peripheral through the read interface (via the unused registers) when the write channel is idle. This clearly contradicts the expected behavior of bus writes, namely that design registers should not be written to during idle cycles.

B. Output Don't Cares

Since the technique in [7] determines signal propagation, it cannot be used to analyze design output signals as they have nowhere to propagate unless the design is embedded in a larger block. To determine if an output signal, o is being assigned a malicious value (used to leak information) when unspecified, we make the observation that a benign assignment to o is one in which o is a static value or retains its previous value. This approach is ③ in Figure 1. When o is unspecified, we expect the output to be gated or remain the same, not update itself based on the values of other design signals.

Our approach to determining if o is static or retains its previous value under $\mathcal{C}$ involves answering 3 satisfiability queries. The formulas analyzed are built using the mined condition, $\mathcal{C}$, along with a formula, f_o, describing the assignment of o. Similar to the approach in [7], f_o is built from the RTL version of the design by using PyVerilog [17] to extract a data-flow graph (DFG) for o, then PySMT [8] is used to build f_o from the DFG and answer the satisfiability queries.

Each output signal, o, is either a *combinational* or *latch*-type signal. If variable o itself appears in the DFG and subsequent formula f_o, o is a latch signal. Figure 4 shows DFGs generated for *overflow* (a combinational signal) and *value* (a latch signal), which appear in a 3-bit counter. The corresponding f_o formulas are given below:

$$\begin{aligned} \textit{overflow} &= (\textit{value} == 7) \ ? \ 1 \ : \ 0 \\ \textit{value} &= rst \ ? \ 0 \ : \ (\textit{enable} \ ? \ \textit{value} + 1 \ : \ \textit{value}) \end{aligned}$$

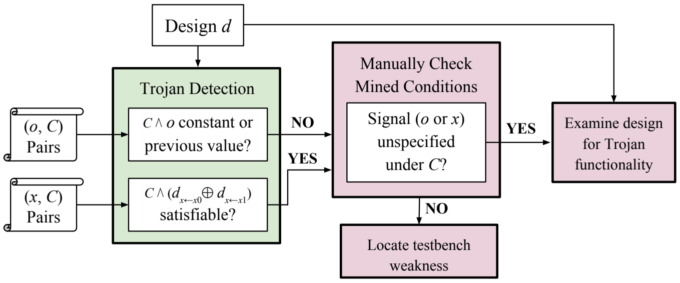
Design d
Trojan Detection
(o, C) Pairs
$C \wedge o$ constant or previous value?
(x, C) Pairs
$C \wedge (d_{x \leftarrow x0} \oplus d_{x \leftarrow x1})$ satisfiable?
NO
YES
Manually Check Mined Conditions
Signal (o or x) unspecified under C?
YES
NO
Examine design for Trojan functionality
Locate testbench weakness

Fig. 5. Trojan Detection Flow

value is a latch signal because it appears in its own DFG (highlighted in blue in Figure 4) and within the formula for *value* itself.

An (o, $\mathcal{C}$) pair is "safe" or Trojan-free if any of the following statements are unsatisfiable, and "dangerous" or Trojan-infected if all queries are satisfiable:

1) $\mathcal{C} \wedge f_o$, if UNSAT f_o is constant 0 under $\mathcal{C}$
2) $\mathcal{C} \wedge \neg f_o$, if UNSAT f_o is constant 1 under $\mathcal{C}$
3) $\mathcal{C} \wedge (f_o \oplus o)$, if UNSAT f_o is equivalent to o under $\mathcal{C}$

Query 3 is only performed if o is a latch signal as it essentially checks if f_o simplifies to o (retains its prior value) under $\mathcal{C}$. If o is a combinational signal, only queries 1 and 2 are necessary to determine if an (o, $\mathcal{C}$) pair is safe or dangerous.

The Trojan leaking the signal *secret data* using *rd_data* when *rd_valid* is 0 in the example bus peripheral shown in Figure 2 would be detected because all 3 of the above queries are satisfiable when o = *rd_data* and $\mathcal{C} = \neg$*rd_valid*. Because of the Trojan circuitry, *rd_data* is not a constant 0 or 1 value when *rd_valid* is 0, nor does it retain prior read values.

Figure 5 summarizes the Trojan detection flow, indicating when manual analysis (the red blocks) is required. It is important to note that human intervention is necessary **only for assertions classified as dangerous**. Safe assertions require no effort on behalf of the designer or verification engineer. (x, $\mathcal{C}$) and (o, $\mathcal{C}$) pairs classified as dangerous need to be manually scrutinized because they indicate one of the following cases is true: 1) the mined condition accurately characterizes when the target signal is unspecified and a Trojan or design bug is present which must be located, or 2) the mined condition actually describes *specified* functionality which was classified as unspecified by the miner because the testbench did not verify this functionality adequately. In either case the design or the testbench requires modification and improvement.

VI. UART Controller Example

The entire mining and Trojan detection flow shown in Figure 1 is used to analyze a Universal Asynchronous Receiver/-Transmitter (UART) circuit from OpenCores [2] designed to interface with a host processor through a Wishbone Bus Interface [3]. Both the original design and a version infected with a Trojan leaking information through the Wishbone bus during idle cycles are analyzed. In the original design, only 20% of the generated assertions are classified as dangerous (requiring

TABLE II
ASSERTION MINING AND TROJAN DETECTION RESULTS FOR UNDETECTED FAULTS IN UART DESIGN (GENERATED USING A DELL OPTIPLEX 7040 WORKSTATION WITH 16GB RAM RUNNING 64-BIT UBUNTU 16.04 LTS)

Outputs Differing	Fault Set	VCD Mining Time (sec)	Assertion Generation Time (sec)	Trojan Detection Time (sec)	Total # Assertions	# Unsafe Assertions
wb_dat_o	1411, 1412, 1413	133.00	0.23	888.89	90	0
	402	3.21	0.04	119.25	24	9
	1042	6.40	0.02	51.80	7	3
int_o	918, 921	26.99	1.05	26.62	367	78
	374, 378, 385	57.10	0.70	1.21	10	8
	1010	6.41	0.75	0.16	1	0
	853, 854, 855, 860, 865, 869, 871, 872	115.16	1.23	0.44	5	1
	849, 850	11.93	0.66	0.21	2	0
	767	3.11	0.05	0.22	1	0
	923, 920, 929	69.10	1.12	1.15	9	7
	895	3.61	0.07	0.20	2	2
	951	3.11	5.54	0.43	1	1

further manual analysis), meaning the remaining 80% are generated and analyzed in a completely automated fashion. Furthermore, all 90 assertions generated for the original design relating to unspecified Wishbone bus functionality are classified as safe, while 53 out of 141 assertions generated for the Trojan-infected design specifically related to the Wishbone bus are classified as dangerous. This shows the ability of our methodology to correctly identify dangerous unspecified functionality in a Trojan-infected version of the design without returning false positives for the same functionality in the original Trojan-free design.

A. Trojan-free Design Analysis

Over 1500 faults are injected into the original UART design using Synopsys Certitude [1] version J-2014.12. An OVM testbench provided by an EDA vendor containing 75 directed testcases and sophisticated design checkers is used to detect the injected faults. Our setup and results for fault injection in the UART core closely follows that of [5]. In this experiment, however, we focus on mining $(o, \mathcal{C})$ pairs, and designate every output in the UART as a target signal.

27 faults are undetected *and* cause UART outputs to differ. The outputs differing are *wb_dat_o*, a 32-bit Wishbone bus signal register data is placed on during a read transaction, and *int_o*, a single bit signal indicating the presence of an interrupt in the UART core. The 27 faults are manually grouped into 12 fault sets (corresponding to the 12 rows in Table II), where faults in a single set affect the same block of code.

For each fault set, trace mining (② in Figure 1) generates $(o, \mathcal{C})$ pairs. The runtime in seconds of interval extraction and ON/OFF-set mining is given by the column labeled "VCD Mining" in Table II. Decision tree construction and assertion extraction time is given by the column labeled "Assertion Generation", and the time to classify all $(o, \mathcal{C})$ pairs as safe or dangerous is given by the column labeled "Trojan Detection." Trace mining and Trojan detection for all faults in Table II takes less than half an hour, illustrating that our technique is certainly scalable to bus peripherals and IP cores with thousands of lines of source code.

The two rightmost columns in Table II give the total number of $(o, \mathcal{C})$ pairs (assertions) generated for each fault set, and how many are classified as unsafe by the Trojan detection technique. It is worth noting that the number of unsafe assertions remains in the single digits for all but one fault set, making manual analysis of each assertion set a reasonable endeavor. Moreover, the engineer no longer has to reason about versions of the design injected with artificial faults as the assertions are mined from the fault-free design. The assertions also condense information across testcases removing the need to manually examine simulation waveforms.

Fault set {918, 921} produces over 300 assertions because the faults are extremely high impact, making *int_o* become X (unknown) for many cycles during almost every testcase. The faults were not detected due to a severe bug in the interrupt checkers which ignores verifying *int_o* when the signal is X. Likely after just a few assertions are examined, the bug would be spotted, the testbench hole closed, and the faults detected, making it unnecessary to analyze the remaining assertions.

Unspecified Wishbone Bus Behavior: The potential of our technique to characterize unspecified functionality is exemplified by the results of fault set {1411, 1412, 1413}. The faults affect when the UART updates values on the Wishbone data bus, which is controlled by output enable signal *oe*. The fault-free assignment for *oe* is given below:

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

wb_dat_o is updated only if a read transaction is occurring, the UART is the selected bus slave, a valid transaction is taking place, and the Wishbone state controller is in State 01. The value of *wb_dat_o* when all of these conditions are not simultaneously met is unspecified because the bus master has no use for bus data when not reading from the UART.

Each fault in the fault set changes the & operator (highlighted in gray) to an | operator, relaxing the conditions under which the value of *wb_dat_o* is updated and causing spurious changes to the signal. These changes go unnoticed by the testbench, which includes a Wishbone protocol checker, because the faults only modify the bus data when it is not being captured during a read transaction and never cause incorrect data to be read from the UART registers.

The assertions generated from fault set $\{1411, 1412, 1413\}$ characterize true unspecified functionality, and since the original UART is Trojan-free, are classified as benign (none of the 90 assertions are dangerous). Manual analysis of these faults and modifications to the testbench are unnecessary because our analysis formally proves no information can leak from the design during idle bus cycles. This is in contrast to the conclusions drawn in [5], where these same 3 faults were flagged, and 1) had to analyzed manually by the designer, and 2) required testbench modification to detect the faults. A major shortcoming of [5] is that all identified unspecified functionality is required to be specified to guarantee the absence of Trojans, whereas our technique uses formal methods to prove the current implementation is secure.

B. Trojan Detection

To demonstrate the ability of our method to detect malicious circuitry, a Trojan leaking information when *wb_dat_o* is unspecified is inserted in the UART. The Trojan does not cause any testcases to fail. We inject faults 1411, 1412, and 1413 in the Trojan-infected UART (they are undetected) and mine (o, $\mathcal{C}$) pairs. Trace mining and formal analysis run-time is similar to the fault-free design.

141 assertions are generated, and 53 are classified as dangerous. Upon examination of a few assertions, the Trojan circuitry is revealed. The increase in total number of assertions results from the increased complexity of the control logic for *wb_dat_o* after Trojan insertion. Since our threat model assumes that no golden model exists, it is critical that unspecified functionality being modified by a Trojan is not classified as benign and specifically highlighted for review.

VII. Conclusion

We address challenges in characterization of unspecified functionality by mining simulation traces produced during mutation testing, and by doing so completely automate the process of detecting Trojans hiding in don't care conditions. Unlike prior methods for identifying unspecified functionality, our technique incorporates formal Trojan detection methods into the characterization flow, removing the need for costly manual analysis of benign functionality. Any functionality highlighted by our technique as dangerous is expressed in the form of assertions which can aid in Trojan localization and testbench debugging. Additionally, we address a blind-spot of existing detection strategies and provide a method capable of identifying malicious behavior under output don't care conditions. We perform the complete trace mining and Trojan detection procedure on Trojan-free and Trojan-infected versions of a UART design showing the ability of our technique to detect malicious modifications to unspecified Wishbone functionality.

VIII. Acknowledgments

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

References

[1] Synopsys certitude: https://www.synopsys.com/verification/simulation/certitude.html.

[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 Spectrum*, 45(5):34–39, May 2008.

[5] N. Fern and K.-T. Cheng. Detecting hardware trojans in unspecified functionality using mutation testing. In *Proceedings of the 2015 International Conference on Computer-Aided Design (ICCAD)*, pages 560–566, 2015.

[6] N. Fern et al. Hiding hardware trojan communication channels in partially specified SoC bus functionality. In *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2016.

[7] N. Fern et al. Detecting hardware trojans in unspecified functionality through solving satisfiability problems. In *Proceedings of the 2017 Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 598–504, 2017.

[8] M. Gario and A. Micheli. PySMT: a solver-agnostic library for fast prototyping of smt-based algorithms. 2015.

[9] M. Hicks et al. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P)*, pages 159–172, 2010.

[10] 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.

[11] W. Li et al. Scalable specification mining for verification and diagnosis. In *Proceedings of the 2010 Design Automation Conference (DAC)*, pages 755–760, 2010.

[12] L. Liu, C.-H. Lin, and S. Vasudevan. Word level feature discovery to enhance quality of assertion mining. In *Proceedings of the 2012 International Conference on Computer-Aided Design (ICCAD)*, pages 210–217, 2012.

[13] S. Mitra, H.-S. P. Wong, and S. Wong. Stopping hardware trojans in their tracks. *IEEE Spectrum*, Jan. 2015.

[14] S. K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data mining and knowledge discovery*, 2(4):345–389, 1998.

[15] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[16] D. Sullivan et al. FIGHT-Metric: Functional identification of gate-level hardware trustworthiness. In *Proceedings of the 2014 Design Automation Conference (DAC)*, pages 173:1–173:4, 2014.

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

[18] S. Vasudevan et al. Goldmine: Automatic assertion generation using data mining and static analysis. In *Proceedings of the 2010 Conference on Design, Automation and Test in Europe (DATE)*, pages 626–629, 2010.

[19] A. Waksman, M. Suozzo, and S. Sethumadhavan. FANCI: Identification of stealthy malicious logic using boolean functional analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, pages 697–708. ACM, 2013.

[20] K. Xiao et al. Hardware trojans: Lessons learned after one decade of research. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(1):6:1–6:23, May 2016.

[21] J. Zhang et al. VeriTrust: Verification for hardware trust. In *Proceedings of the 2013 Design Automation Conference (DAC)*, pages 61:1–61:8, 2013.