Evaluating Assertion Set Completeness to Expose Hardware Trojans and Verification Blindspots

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

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

***Abstract*—Assertion-based verification has been adopted by industry as an efficient specification mechanism. Handwritten assertions encode design intent in a parsable format and have been traditionally used to verify an implementation *conforms* to the properties outlined by the assertions. Our work makes the observation that design behavior *not* covered by the assertion set is equally revealing and can be leveraged to identify malicious behavior (hardware Trojans) as well as verification blindspots. The difficulty in examining this unspecified and unverified behavior is differentiating between benign functionality that is truly don't care and that which leaks information or violates design intent. Prior work exploring assertion set completeness suffers from this inability to distinguish benign unspecified functionality from actual verification holes, while existing Trojan detection techniques can differentiate these categories, but require unspecified functionality already be characterized. Our technique uses the assertion set and simulation trace data available in most industry design flows to characterize unspecified functionality then separates Trojans and verification blindspots from benign behavior using existing Trojan detection methods. Using our technique, we uncover missing functionality in a first-in first-out (FIFO) queue implementation and demonstrate detection of information leakage Trojans. We also illustrate Trojan detection for a system containing several components connected by an AXI4-Lite bus by analyzing the completeness of the AXI4-Lite assertion set provided by ARM.**

I. Introduction

Two import questions in verification that have been addressed for decades are: 1) "how well does the set of assertions or properties capture the design intent?" and 2) "how thorough is the testbench in verifying the implementation doesn't violate these properties?" As the complexity of design, manufacturing, testing, and distribution of silicon chips increases, there have been concerns about security and trustworthiness [1]. One such threat is the inclusion of malicious design modifications, known as hardware Trojans. Trojans can be inserted pre-silicon by any engineers, design tools, and software programs with access to the RTL code base or design netlist, and during/after fabrication by anyone involved in the chip manufacturing and testing process and can implement attacks ranging from denial of service to stealthy information leakage mechanisms [2]. It is now necessary to answer a third question during pre-silicon verification: 3) "does my design contain extra malicious functionality in addition to core design functionality?"

Assertion-based verification (ABV) [3] is widely used in industry as a key component in the pre-silicon functional verification infrastructure. Because ever increasing design complexity often makes formal property verification intractable, simulation-based methods have become the workhorse of pre-silicon verification in industry, and commercial RTL simulation tools support property specification languages such as SystemVerilog Assertions (SVA) [4].

Gauging the completeness and quality of the specification (the first verification question) has been addressed by techniques which use formal analysis of the property set and in some cases the design implementation as well to determine the fraction of behavior not covered by the specification [5], [6], [7], [8], [9], [10], [11]. For simulation-based techniques, the second verification question is addressed by structural and functional coverage metrics [12], which estimate the amount of design functionality activated by the test cases, and mutation testing [13], which gauges the ability of the testbench to propagate and detect errors.

One salient point is that all metrics and methods proposed to qualify specification completeness and testbench effectiveness struggle with addressing *unspecified functionality* and answering the third verification question. For example, when [11] is used to determine the completeness of properties given in the AMBA AHB on-chip bus protocol specification, the coverage is very low, and it is concluded that a majority of functionality is left unspecified on purpose to allow freedom in the hardware implementation of the bus protocol. There is no mechanism provided to identify only underspecification masking potential bugs which is the limitation this work overcomes.

In modern designs, it is not practical to enumerate behavior for every signal at every cycle, however Trojans hiding in RTL don't cares and modifying existing on-chip bus infrastructure to leak information during idle bus cycles [14] have been proposed. The technique provided in [15] can detect these Trojans before tape-out, but only after unspecified design functionality is characterized. Property set completeness analysis and [15] are symbiotic because the set of design conditions not covered by the property set can be used as input to [15], which is capable of classifying these conditions as either benign unspecified functionality or Trojan-infected functionality. This work is the first to recognize the power of combining these two techniques. Because SystemVerilog Assertions are tightly coupled with testbench code, and contain constructs not easily handled by formal methods, we provide a novel simulation-based assertion set completeness analysis technique instead of employing an existing formal one.

Figure 1 provides an overview of our technique, showing how our proposed simulation trace mining methodology

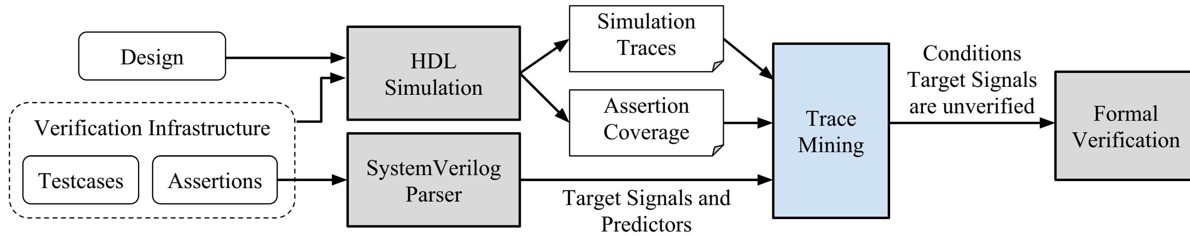

Fig. 1. Simulation-based Assertion Set Analysis Methodology Overview

(shown as a blue square box) can be combined with existing tools and techniques (shown using gray square boxes) to highlight verification holes and dangerous unspecified functionality. Our mining technique produces a list of conditions specifying when a signal is *not* verified by the assertion set. These unverified conditions fall under 3 categories:

1) **Specification/Verification Hole:** under the unverified condition the target signal is important to the core functionality of the design, and the assertion set needs to be expanded to cover this missing case
2) **Malicious Unspecified Functionality:** under the unverified condition the target signal has no relevance to the design, but contains a Trojan used to leak information
3) **Benign Unspecified Functionality:** under the unverified condition the target signal is a don't care and does not contain a Trojan meaning no modifications to the verification infrastructure are necessary

The conditions in Categories 1 and 2 must be addressed by the verification team, however expending effort to interpret and develop additional assertions and testcases for truly benign unspecified functionality (Category 3) is an unnecessary waste of resources. To help distinguish between these categories, the unverified conditions for each target signal are passed to the formal Trojan detection technique proposed in [15].

The key contributions of our work are:

1) Combining specification coverage analysis with formal Trojan detection methods to highlight specification/verification holes and Trojans in unspecified functionality
2) A novel simulation-based assertion set completeness analysis technique
3) Demonstration of our technique on the AMBA AXI4-Lite bus protocol assertion set provided by ARM

The rest of the paper is structured as follows: Section II reviews related work and provides our threat model, Section III presents our assertion set analysis technique, Section IV walks through our technique and its application to Trojan detection for a simple FIFO circuit, Section V applies our technique to a design utilizing the industry standard AXI4-Lite bus protocol, and in Section VI we conclude.

II. Related Work and Threat Model

Assertion mining techniques such as Goldmine [16] and Scalable Assertion Miner (SAM) [17] analyze simulation trace waveforms to extract a set of assertions which can be scrutinized to detect anomalous behavior. These techniques describe the design as implemented, so it is ultimately up to a human engineer to determine if the assertions describe correct behavior or reflect erroneous behavior. Moreover, these techniques create new assertions and do not quantify the completeness of the generated assertion set.

The closest related work is [18], which mines simulation traces produced during mutation testing to find conditions under which the fault-free and faulty designs differ for all *undetected* faults. Because undetected faults correspond to changes in the design the testbench is blind to, the generated conditions describe unverified design functionality. The main applications of [18], like this work, are identification of verification holes and hardware Trojans in unspecified functionality, but [18] uses the testbench's ability to detect mutants to locate verification blind spots while our method uses assertion coverage to extract this information.

Mutation testing [13] makes the technique proposed in [18] computationally intensive because the entire regression test suite must be run and the generated traces mined for every single mutated design with an injected fault (of which there are typically hundreds). Our method can find unverified design conditions *with the equivalent complexity of analyzing a single mutated/faulty design* because unlike [18] we take advantage of the information about design intent coded into assertion sets, available for most industry designs. This means our method can provide the same insight as [18] with orders of magnitude less simulation time, however if assertion-based verification is not used then [18] can still be applied, whereas our technique can not.

Threat Model: Like [18], our technique analyzes a pre-silicon design for bugs and Trojans, and requires no Trojan-free model (rather the goal of our technique is to increase confidence that the HDL design *is* a golden model). We assume the assertion set and verification infrastructure is Trojan-free, but not necessarily bug-free.

III. Assertion Set Analysis Technique

Our technique requires the verification infrastructure to contain a set of assertions which take the basic form of:

$$antecedent\ sequence\ (AS) \rightarrow consequence\ sequence\ (CS)$$

The antecedent sequence is a design condition which must be met before checking if the consequence sequence is satisfied. If the consequence sequence is not satisfied then the assertion is violated and an error is raised. The *AS* can be set to *True* if the property has no precondition.

Assertion Coverage: Assertion coverage, a feature provided by all commercial RTL simulators, tracks how many times the antecedent sequence is satisfied during simulation for each assertion. While it can reveal aspects of the design described by the assertions but never exercised during simulation, it can't reveal functionality missing from the assertion set itself. Finding design behavior currently not covered by the assertion set is the goal of our simulation trace mining technique, and we use the information provided by assertion coverage along with simulation traces to help reveal these verification blindspots.

Mining produces a list of conditions describing when a signal is *not* verified by the assertion set. Because our analysis relies on simulation traces, the quality of the test cases can impact the effectiveness of our technique. This is a drawback of all simulation-based (vs. formal) analysis, therefore it is recommended that 100% assertion coverage is achieved and the test suite is mature before applying our technique.

A. Mining Unverified Conditions

The trace mining technique detailed in this section is shown as a blue box in Figure 1. The design and verification infrastructure, including the assertions and tests, is used to simulate the design and produce traces recording testbench and design signal values at every simulation cycle along with an assertion coverage report. These are processed by our mining technique to produce a list of conditions describing when design signals are not covered by any assertions. Mining consists of 3 main steps: target signal and predictor identification, function mining, and function compression. The following terms are used in the subsequent description of the mining procedure:

- *Assertion Set:* set of assertions describing behavior of signals in the design
- *Target Signal Set:* set of signals appearing in a consequence sequence in any assertion in the assertion set

For a particular target signal, s, from the target signal set:

- *Antecedent Sequence Set:* set of antecedent sequences where s appears in the consequence sequence
- *Predictor Set* {*p0, p1, ...*}*:* set of signals appearing in the antecedent sequence set for s

1) Target Signal and Predictor Identification: The first step in assertion set analysis is identifying the target signal set. Automation of this step as well as identification of the predictor set requires a SystemVerilog parser capable of interpreting the full assertion syntax. While commercial RTL simulation tools can parse and interpret advanced features of the SystemVerilog language, the resulting syntax tree data structures used by these tools are not easily accessible so in this work we manually identify the target signals and predictors, but there is no conceptual reason why this process can't be easily automated.

If an antecedent sequence spans more than a single cycle, the mined conditions for the target signal will also be sequential. To handle this, the predictors in a sequential antecedent sequence will be unrolled for the number of cycles the sequence spans. As an example, the following assertion states a flag signal should be set when the design transitions from the IDLE to ACTIVE state:

```
flag_check: assert property @(posedge clk)
    (state==IDLE) ##1 (state==ACTIVE) |-> flag;
```

In this example *state* is a 2-bit signal which can be assigned IDLE, ACTIVE, or ERROR. The target signal is *flag*, and since the antecedent sequence spans 2 cycles and *state* is 2 bits, the predictor set is {*state*[1] (t-1), *state*[0] (t-1), *state*[1], *state*[0]}.

The assertion coverage log file contains the information necessary to determine the number of cycles a predictor should be unrolled. Below is an example of assertion coverage information given by Synopsys VCS:

```
flag_check: started at 20ns succeeded at 30ns
flag_check: started at 50ns succeeded at 60ns
```

If the clock period is 10ns then it is easy to conclude both the current and previous value of *state* are required in the antecedent sequence, and that *state* needs to be unrolled one cycle. If the antecedent sequence can be satisfied by a number of different possible traces with varying sequential depth, but not all trace lengths are seen during simulation, there is potential to underestimate the sequential depth of the AS because we are using a simulation-based technique and relying on the completeness of the tests to activate the assertions.

2) Function Mining Using Coverage Intervals: The goal of our technique is to mine simulation traces to build a function $\mathcal{C}$. When $\mathcal{C} = 0$ the target signal is not verified by the assertion set, and when $\mathcal{C} = 1$ the target signal is covered. Predictors are the function variables, and mining effectively builds the truth table for $\mathcal{C}$. A new truth table row is mined at each simulation cycle using the values of the predictors at the current cycle (time t) and previous ($t - n$) cycles if the predictors are unrolled. The function output is 1 (the row is added to the ON-set of $\mathcal{C}$) if during the current simulation cycle any assertions verifying the target signal are covered, and 0 (row added to the OFF-set) otherwise.

For example, since the `flag_check` assertion is covered at 30ns and 60ns, {*state*@20ns, *state*@30ns} and {*state*@50ns, *state*@60ns} are added to the ON-set, and the predictor values at all other cycles are added to the OFF-set. Since transitioning from IDLE to ACTIVE is the only condition which activates the assertion {IDLE, ACTIVE} will be the only row in the ON-set. The design is likely to traverse other paths in the state machine during simulation (e.g. {ACTIVE, IDLE} or {IDLE, ERROR}), and these will be added to the OFF-set. After mining, $\mathcal{C}$ will reveal that *flag* is unverified for many of the state transitions seen during simulation. For example, a bug in the design could set *flag* on the first transition from IDLE to ACTIVE, but forget to unset the flag when transitioning from ACTIVE to IDLE. With the current assertion set, the bug would never be found as the assertion would always succeed.

3) Compression Using Decision Trees: Each row in the OFF-set represents an unverified condition, and while all rows in the OFF-set can be passed to the formal analysis technique, when the number of predictors is large it is crucial to further compress the function using decision trees [19]. Decision trees are a machine learning classifier, and for our application the

two classes are $\mathcal{C} = 0$ and $\mathcal{C} = 1$, and the classifier input is the predictor set. Using the ON-set and OFF-set rows as training data, the classifier is constructed to maximally separate the 2 classes using the smallest number of predictors. Our use of decision trees to compress simulation trace mining results is similar to [16] and [18], and due to space considerations we refer the reader to these works for further details.

B. Formal Analysis

For each target signal, trace mining produces a list of conditions during which the signal is unverified. While these conditions can be manually analyzed, it is worthwhile to perform further automated analysis to formally prove the target signal under the unspecified condition, $\mathcal{C}$, is completely benign. Any conditions proven benign do not need to be manually inspected or addressed by revising the design and verification infrastructure, greatly reducing the cost of our technique. The criteria for benign behavior depends on if the target signal is an input/internal signal or a primary output.

If an input/internal signal, x, under unverified condition $\mathcal{C}$, propagates to design outputs under $\mathcal{C}$, x is certainly not benign. The technique proposed in [15] determines the satisfiability of $\mathcal{C} \wedge (o_{x\to x_0} \oplus o_{x\to x_1})$ for all outputs in the design. If satisfiable, there exists a pair of different values for x (x_0 and x_1) which cause differences in the output, o, under $\mathcal{C}$ meaning x should be further examined under this condition for verification blindspots, bugs, or hardware Trojans.

Target signals which are primary outputs are considered benign if under $\mathcal{C}$ the signal is gated (assigned a static 0 or 1) or retains its previous value, and [18] provides a mechanism to determine this by answering 3 satisfiability queries, where o is the target signal variable and f_o is the corresponding formula:

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 $\mathcal{C}$

These satisfiability queries can be answered using a variety of SAT and SMT solvers, but in this work we employ PyVerilog [20] to extract a data-flow graph (DFG) for all design outputs, then PySMT [21] is used to build SMT formulas from the DFGs and answer the satisfiability queries. Sequential behavior can be analyzed by unrolling the SMT formulas.

IV. FIFO Example

We demonstrate our technique in detail using a simple FIFO design based on code available online [22]. The original design, shown in black in Figure 2, is a synchronous FIFO with a data width of 8 bits and a depth of 16. Trojan circuitry, which routes *data_in* directly to *data_out* when *data_out* is idle (no read occurring), is shown in red. The chip select (*cs*) and enable (*en*) control signals dictate when data is written into or read out of the FIFO. Both *wr_cs* and *wr_en* must be set to write data, and both *rd_cs* and *rd_en* must be set to read data from the FIFO. The inserted Trojan circuitry hides behind the unspecified and unverified condition when only one of the two control signals are set.

Using our technique we identify functionality missing from the FIFO design implementation (a design bug) as well as missing from the assertion set (verification blindspot). We also insert a Trojan leaking information using the *data_out* signal when it is unspecified (more details provided in Section IV-C) which is successfully detected by our method.

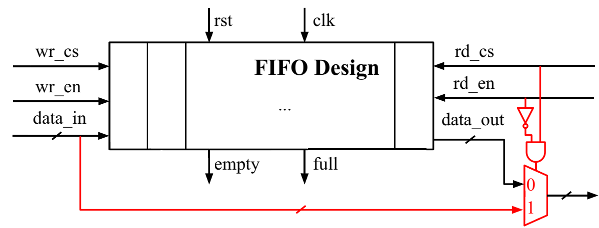

Fig. 2. Block Diagram of FIFO Example (Trojan Circuitry Shown in Red)

A. The Assertion Set

We develop a set of assertions using SystemVerilog which describe the expected behavior of key signals in the FIFO design. The two assertions below describe the behavior for the *empty* signal. When the design is not in reset, *empty* should be set when the FIFO contains no items (the variable *q_size* gives the current number of items in the FIFO), and not set when there are items stored. When mining conditions *empty* is unverified the predictor signals are *rst* and *q_size*.

```
empty_set: assert property (@(posedge clk)
    ~rst && (q_size == 0) |-> empty);
empty_unset: assert property (@(posedge clk)
    ~rst && (q_size > 0) |-> ~empty);
```

The assertions for the *full* signal are similar and not shown due to space constraints. When the design is not in reset, *full* should only be set when the number of items in the FIFO is equivalent to the maximum number of elements the FIFO can hold. The predictors are also *rst* and *q_size*.

The `read_check` assertion (given below) checks the correctness of the *data_out* signal during a read operation. In the testbench there is a software model of the FIFO which is used to generate the *read_data_expected* signal. The predictors used to mine unverified conditions are *rst*, *rd_cs*, *rd_en*, and *q_size*.

```
read_check: assert property (@(posedge clk)
    ~rst && rd_cs && rd_en && (q_size > 0) |=>
      data_out == read_data_expected);
```

B. Finding Design and Testbench Bugs

In this assertion set there are 3 target signals: *empty*, *full*, and *data_out*. Simulation trace mining produces a set of conditions describing when each of the target signals is unverified. For *empty* and *full*, a single condition, *rst*, is returned indicating that the value of these target signals is not checked during reset. While this is a gap in verification coverage, further formal analysis employing the satisfiability queries outlined in Section III-B shows that in the design implementation *empty* and *full* are assigned 1 and 0 respectively as long as the reset signal is held meaning no malicious information leakage is possible during reset using these signals. For the *data_out* signal, trace mining results in 4 unverified conditions: 1)

$\neg rd_cs$, 2) $rst \wedge rd_cs$, 3) $\neg rst \wedge rd_cs \wedge \neg rd_en$, and 4) $\neg rst \wedge rd_cs \wedge rd_en \wedge (q_size == 0)$.

Each of these conditions is analyzed using the satisfiability queries outlined in Section III-B. Conditions 1-3 are classified as *safe* by the formal analysis tool, meaning *data_out* is either static or retains its prior value while the conditions hold. Condition 4 is classified as *dangerous*, which was initially surprising because when $q_size == 0$ no read operation should occur. The satisfying input assignment returned by the formal analysis tool included $wr_cs == 1$ and $wr_en == 1$, which reveals a missing corner case: reading and writing to the FIFO simultaneously when the FIFO is currently empty.

This corner case is missing from both the verification plan as well as the design itself. Simultaneous read and write operations do not cause an issue in a non-empty FIFO as valid data is read out the front of the queue and new data is pushed into the back, but when the FIFO is empty random data is read out even if valid data is simultaneously being written. The correct behavior in this case is for *data_in* to be passed directly to *data_out*. Our assertion set analysis method was able to catch this erroneous functionality, leading to revisions in both the design and testbench to cover this missing case.

C. Detecting Hardware Trojans

We insert the Trojan circuity shown in red in Figure 2. This circuitry routes *data_in* directly to *data_out* when $rd_en == 0$ and $rd_cs == 1$. Normally during this condition *data_out* retains its prior value because no read operation is taking place, but with the Trojan circuitry any data currently driving the *data_in* lines becomes immediately visible at *data_out*. If the FIFO is embedded in a larger design where sensitive data flows through the FIFO this shortcut may result in information leakage not possible under the original design.

The analysis results for the *empty* and *full* signals match those in the previous section as the Trojan doesn't affect these signals. For the *data_out* signal several mined conditions are classified as dangerous after formal analysis and the satisfying input assignment returned contains the condition for Trojan activation, $rd_en == 0 \wedge rd_cs == 1$, illustrating the potential of our technique to highlight malicious functionality.

V. AXI4-Lite Assertion Set Analysis

We analyze the assertion set provided by ARM [23] for the AMBA AXI4-Lite bus protocol [24] to identify unverified conditions related to bus interconnect signals in the system shown in Figure 3. The bus masters are bus functional models (BFMs) written in SystemVerilog used to drive transactions on the interconnect [25], while the interconnect fabric [26] and bus slaves are implemented in Verilog. The bus slaves are simple 8-bit adders whose operands and resulting computation are accessed through register read and write operations controlled by AXI4-Lite signals as shown in Figure 4.

The Assertion Set: There are 48 assertions provided by ARM [23] which check the compliance of each of the 5 AXI4-Lite interfaces. During simulation of both the Trojan-infected and Trojan-free designs none of the assertions are violated.

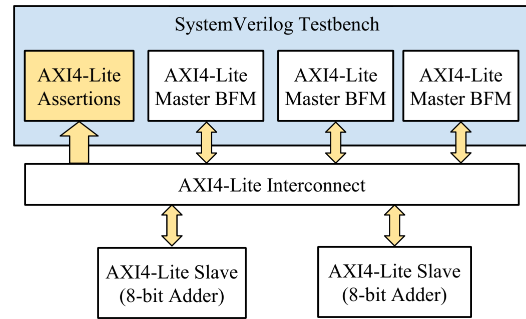

Fig. 3. AXI4-Lite System with Compliance Checking Assertions

A. Trojan-free Design Analysis

Our technique is used to mine unverified conditions for the 17 AXI4-Lite bus interface signals shown in Figure 4. Mining the simulation traces takes 3.9 seconds, and filtering the resulting unverified conditions using formal methods for all target signals takes 1.4 seconds. There are a total of 31 unspecified/unverified conditions mined by our technique. In the Trojan-free design 27 out of 31 conditions are benign, however 4 conditions are classified as dangerous.

The dangerous conditions for ARVALID (the read address valid signal) and ARADDR (the read address) correspond to when the design is in reset (reset_n is active low). It is unexpected that outputs can change based on different values of ARVALID and ARADDR while the design is in reset. The output affected is RDATA (read data), and upon examination of the read operation state machine it is revealed that when in the "read idle" state, if ARVALID is set, the address given by ARADDR is accessed and RDATA updated. When reset is active, the state machine is forced to remain in the "read idle" state, therefore if the reset signal to the coprocessor is held, but the ARVALID and ARADDR signals fluctuate, data from different register locations appears at RDATA. RDATA is a 32 bit signal, but the coprocessor uses a maximum of 8 bits to implement registers. During reset unused bits are not zeroed, and for this reason our analysis identifies this as a potential source of information leakage as a Trojan could assign these bits any value including secret internal signals.

The dangerous conditions for RDATA ($\neg$RREADY and RREADY $\wedge$ $\neg$RVALID) are a true false positive as our formal analysis is detecting RDATA changing based on combinational logic enabled once a valid read address appears, which is one cycle in advance of setting the RVALID signal. Because the assertions only check RDATA when RVALID is set, the previous cycle appears to be an unverified condition under which an output signal can change.

B. Trojan Detection

Our technique successfully highlights bus signals used to implement a Trojan which writes data to unused register bits in the adder coprocessor. Figure 4 shows how the adder coprocessor registers are accessed using AXI4-Lite bus signals. The addressing scheme allows registers to be 32 bits, however the maximum register size used by the coprocessor is 8 bits.

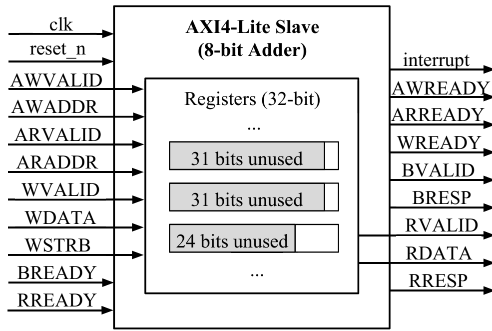

Fig. 4. 8-bit Adder Coprocessor with an AXI4-Lite Bus Interface

Unused space can be used to hide Trojan data and allow covert communication between two malicious bus masters who can access the adder but are unable to communicate directly with each other using the existing bus topology.

To make the communication covert and prevent other bus masters from accidentally zeroing the unused bits selected for Trojan communication, existing write channel signals are still used to place information into these bits, but not using the mechanism outlined in the AXI4-Lite protocol. Instead, writing to the unused register bits occurs only when the write channel valid signal (WVALID) is LOW and the 4-bit write strobe signal (WSTRB) is set to all ones. This particular strobe value does not naturally occur when accessing the coprocessor as there are no 32-bit registers. The malicious data stored in unused register bits can then be read out by another malicious bus component using a normal read transaction.

In the Trojan-free analysis results no conditions are classified as dangerous for the WDATA and WSTRB signals, meaning that these signals can't propagate under unverified conditions to design outputs. When the analysis is re-run for the Trojan-infected design, conditions requiring WVALID == 0 are classified as dangerous. Input assignments proving propagation of WDATA and WSTRB to the read data output under unverified conditions were given by the SMT solver. Included was the assignment of 4'b1111 to WSTRB, which precisely highlights the condition for writing Trojan data into the coprocessor registers.

VI. Conclusion

Determining the quality and completeness of the verification infrastructure, including design assertions, is a challenging but important problem which is further complicated by the potential inclusion of extra malicious functionality hiding in unspecified design behavior. Existing testbench and assertion set completeness metrics do not handle unspecified functionality securely, and existing Trojan detection techniques targeting analysis of unspecified behavior miss the opportunity to use the assertion set to characterize unspecified and unverified conditions. We overcome both of these shortcomings by mining simulation trace data and assertion coverage to find unverified conditions then use formal methods to differentiate benign unspecified behavior from verification blindspots and malicious functionality. We analyze a FIFO design with a simple assertion set and a system consisting of several components which communicate using the AXI4-Lite bus protocol monitored by over 40 compliance checking assertions. Our method successfully uncovers verification blindspots masking actual design bugs and potentially malicious behavior and detects information leakage Trojans inserted in both designs.

VII. Acknowledgements

This work was supported by NSF/SRC STARSS (1526695) and RGC of the Hong Kong SAR, China (HKUST 16207917).

References

[1] M. Rostami, F. Koushanfar, and R. Karri, “A primer on hardware security: Models, methods, and metrics,” *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1283–1295, 2014.
[2] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, “Hardware trojans: lessons learned after one decade of research,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 1, pp. 6:1–6:23, May 2016.
[3] H. D. Foster, A. C. Krolnik, and D. J. Lacey, *Assertion-based design*. Springer Science & Business Media, 2004.
[4] “IEEE standard for SystemVerilog–unified hardware design, specification, and verification language,” *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, February 2018.
[5] O. Kupferman *et al.*, “A theory of mutations with applications to vacuity, coverage, and fault tolerance,” in *FMCAD*, 2008, pp. 1–9.
[6] H. Chao, H. Li, T. Wang, X. Li, and B. Liu, “An accurate algorithm for computing mutation coverage in model checking,” in *ITC*, 2016.
[7] H. Chockler, D. Kroening, and M. Purandare, “Coverage in interpolation-based model checking,” in *DAC*, June 2010, pp. 182–187.
[8] F. Haedicke, D. Große, and R. Drechsler, “A guiding coverage metric for formal verification,” in *DATE*, 2012, pp. 617–622.
[9] D. Große, U. Kuhne, and R. Drechsler, “Estimating functional coverage in bounded model checking,” in *DATE*, 2007.
[10] J. Bormann *et al.*, “Complete formal verification of tricore2 and other processors,” in *DVCon*, 2007.
[11] M. Oberkonig *et al.*, “A quantitative completeness analysis for property-sets,” in *FMCAD*, 2007, pp. 158–161.
[12] S. Tasiran and K. Keutzer, “Coverage metrics for functional validation of hardware designs,” *IEEE Design & Test of Computers*, vol. 18, no. 4, pp. 36–45, July 2001.
[13] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.
[14] N. Fern *et al.*, “Hiding hardware trojan communication channels in partially specified SoC bus functionality,” in *TCAD*, 2016.
[15] N. Fern, I. San, and K.-T. Cheng, “Detecting hardware trojans in unspecified functionality through solving satisfiability problems,” in *ASP-DAC*, 2017, pp. 598–504.
[16] S. Vasudevan *et al.*, “Goldmine: Automatic assertion generation using data mining and static analysis,” in *DATE*, 2010, pp. 626–629.
[17] W. Li *et al.*, “Scalable specification mining for verification and diagnosis,” in *DAC*, 2010, pp. 755–760.
[18] N. Fern and K. T. Cheng, “Mining mutation testing simulation traces for security and testbench debugging,” in *ICCAD*, 2017, pp. 714–721.
[19] J. R. Quinlan, “Induction of decision trees,” *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.
[20] S. Takamaeda-Yamazaki, “Pyverilog: A python-based hardware design processing toolkit for verilog hdl,” in *Applied Reconfigurable Computing*, 2015, pp. 451–460.
[21] M. Gario and A. Micheli, “PySMT: a solver-agnostic library for fast prototyping of smt-based algorithms,” 2015.
[22] D. K. Tala, “Synchronous fifo.” [Online]. Available: http://www.asic-world.com/examples/systemverilog/syn_fifo.html
[23] “AMBA 4 AXI4, AXI4-Lite and AXI4-Stream Protocol Assertions BP063 Release Note (r0p1-00rel0),” ARM. [Online]. Available: https://silver.arm.com/browse/BP063
[24] *AMBA AXI and ACE Protocol Specification, Issue E*, ARM, 2013.
[25] “AXI4 BFM.” [Online]. Available: https://github.com/sjaeckel/axi-bfm
[26] *DS768: LogiCORE IP AXI Interconnect (v1.02.a)*, Xilinx Inc., 2011.