

# Evaluating Security Requirements in a General-Purpose Processor by Combining Assertion Checkers with Code Coverage

Michael Bilzor\*, Ted Huffmire†, Cynthia Irvine†, Tim Levin†  
 \*U.S. Naval Academy †U.S. Naval Postgraduate School

**Abstract**—The problem of malicious inclusions in hardware is an emerging threat, and detecting them is a difficult challenge. In this research, we enhance an existing method for creating assertion-based dynamic checkers, and demonstrate how behavioral security requirements can be derived from a processor’s architectural specification, then converted into security checkers that are part of the processor’s design.

The novel contributions of this research are:

- We demonstrate the method using a set of assertions, derived from the architectural specification, on a full-scale open-source general-purpose processor design, called OpenRISC. Previous work used only a single assertion on a toy processor design.
- We demonstrate the use of our checker-generator tool, called `psl2hdl`, which was created for this research.
- We illustrate how the method can be used in concert with code coverage techniques, to either detect malicious inclusions or greatly narrow the search for malicious inclusions that use rare-event triggers.

## I. MALICIOUS INCLUSIONS

The threat of subversions in processors has gained attention over the last few years, as noted in government reports [1], academic research [2] and the media [3].

A malicious inclusion (MI)<sup>1</sup> is a mechanism implanted in a processor by which an attacker circumvents a processor’s normal functionality. MIs can get into a hardware design in a number of ways, and at different stages of design: an attacker can try to compromise a processor’s high-level design, low-level design, or even make physical modifications after it is manufactured. Wang, Tehranipoor, and Plusquellic give a taxonomy of malicious inclusions and examine the challenge of detecting them [4].

Hardware security, in particular the detection of MIs in processors, is a difficult challenge. Most detection efforts to date have focused on equivalence-checking methods, in which one processor or processor design is compared with another [5], [6]. Design analysis methods to date have focused on identifying unused or rarely-used circuits [7].

The method we use for detecting MIs, originally presented by Bilzor et al. [8], takes a different approach; it is based on the following observations:

- A security policy describes behaviors that are either *permitted* or *prohibited* [9].
- The design of a general-purpose processor is usually based on some governing document, called an architectural specification, containing descriptions of permitted

and prohibited behaviors, which can be modeled using assertions.

- In the published examples of MIs to date, the MIs often appear to cause a processor to express behaviors that are prohibited by the architectural specification. Therefore, *the action of some MIs might be detectable at runtime as violations of synthesized assertion-checkers*, evaluating those behavioral restrictions.
- Hardware engineers already use assertions to establish the functional correctness of designs [10]; it seems natural to also use assertions, which describe behaviors, to more generally identify permitted and prohibited behaviors.
- Assertions, which derive from temporal logic, can be converted into synthesizable *checkers* – hardware design units which model the evaluation of the assertion formula over time against a set of input values, and can be made part of the design units being checked [11].

## II. MODELING BEHAVIORAL RESTRICTIONS

### A. Methodology and Workflow

Bilzor et al. introduced a workflow for generating PSL-based runtime security checkers [8], but that demonstration included only a single checker, used a very simple processor model, and did not include coverage analysis, or employ a fully developed checker generator tool. The steps of the workflow are:

- In the processor architecture, identify statements which specify behaviors that are permitted or prohibited.
- Translate the text statements into PSL assertions, using the appropriate logic signals from the implementation.
- Using the available automated tools, translate the assertions into synthesizable HDL modules, called *checkers*.
- Attach the checkers to the design, and evaluate them under simulation or FPGA emulation, using a testbench.
- If desired, leave the checkers in the design, and fabricate them along with the processor, to facilitate detection of MIs in real time.

We follow this basic workflow in our demonstration, using our new checker-generator, and also show how it can be used in conjunction with code coverage to narrow the search for obscure MIs and triggers.

<sup>1</sup>Sometimes referred to as a Hardware Trojan.

## Report Documentation Page

*Form Approved*  
*OMB No. 0704-0188*

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE <b>JUN 2012</b>	2. REPORT TYPE	3. DATES COVERED <b>00-00-2012 to 00-00-2012</b>			
4. TITLE AND SUBTITLE <b>Evaluating Security Requirements in a General-Purpose Processor by Combining Assertion Checkers with Code Coverage</b>		5a. CONTRACT NUMBER			
		5b. GRANT NUMBER			
		5c. PROGRAM ELEMENT NUMBER			
6. AUTHOR(S)		5d. PROJECT NUMBER			
		5e. TASK NUMBER			
		5f. WORK UNIT NUMBER			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) <b>Naval Postgraduate School, Monterey, CA, 93943</b>		8. PERFORMING ORGANIZATION REPORT NUMBER			
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)			
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)			
12. DISTRIBUTION/AVAILABILITY STATEMENT <b>Approved for public release; distribution unlimited</b>					
13. SUPPLEMENTARY NOTES <b>Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), San Francisco, CA, June 2012, pp. 49-54</b>					
14. ABSTRACT <b>The problem of malicious inclusions in hardware is an emerging threat, and detecting them is a difficult challenge. In this research, we enhance an existing method for creating assertion-based dynamic checkers, and demonstrate how behavioral security requirements can be derived from a processor's architectural specification, then converted into security checkers that are part of the processor's design. The novel contributions of this research are - We demonstrate the method using a set of assertions, derived from the architectural specification, on a full-scale open-source general-purpose processor design, called OpenRISC. Previous work used only a single assertion on a toy processor design. - We demonstrate the use of our checker-generator tool, called psl2hdl, which was created for this research. - We illustrate how the method can be used in concert with code coverage techniques, to either detect malicious inclusions or greatly narrow the search for malicious inclusions that use rare-event triggers.</b>					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT <b>unclassified</b>	b. ABSTRACT <b>unclassified</b>	c. THIS PAGE <b>unclassified</b>	<b>Same as Report (SAR)</b>	<b>6</b>	

Table I  
Comparison of selected features of PSL checker generators.

Feature	SynPSL	MBAC	psl2hdl
Parse All of PSL Simple Subset	✓	✓	✓
Implement All Simple Subset Assertions		✓	✓
Create Synthesizable Checkers	✓	✓	✓
Implement Ranged SEREs	✓	✓	✓
Generate Abstract Syntax Tree Output			✓
Parse All of Underlying Flavor (Verilog)		✓	
Support Built-in Function Calls		some	some
DFA Minimization		partial	✓
Full Boolean-Layer Optimization			✓
VHDL and Verilog Output			✓

### B. The Property Specification Language

The Property Specification Language (PSL) is a temporal logic language, designed to describe the behavior of electronic circuits over time [12]. PSL is primarily used in functional verification of processor designs in hardware design language (HDL) format, such as VHDL or Verilog. For a detailed look at PSL, the reader is referred to the excellent summary by Eisner and Fisman [13].

### C. A “Checker Generator” - psl2hdl

For this research, we created our own tool for synthesizing PSL-based assertion checkers. Written in Python, our tool follows the example of Boulé and Zilic, who developed a checker-generator method for PSL; their checker generator is called MBAC [11], [14]. We started with a PSL lexer and parser, based on the Python parsing add-on P-L-Y [15], constructed by Findenig for the SynPSL checker-generator [16], and then built our own full checker-generator, called psl2hdl. We added several new features, such as the construction and display of PSL abstract syntax trees in Graphviz format [17], addition of complex boolean functions and boolean simplifications, and support for both VHDL and Verilog output. Because the method uses automata as an intermediate representation, we also implemented a full state-minimization algorithm.

The principal advantage of synthesizable checkers is that they can be used across essentially the entire processor development cycle: simulation, FPGA emulation, and silicon fabrication. Some commercial tools support PSL assertions in simulation, but using synthesizable checkers instead makes them supportable by any HDL simulator. The minor enhancements added to psl2hdl are compared with other checker-generators in Table I.

## III. DETECTING PROCESSOR MALICIOUS INCLUSIONS

The idea of the experiment is to illustrate how processor MIs can be detected at runtime. Because we designed both the checkers and the MIs, the demonstration is academic in that regard. We are not aware of any *adversarial* experiments in MI detection using general-purpose processors, where one group designs the MIs, and another group attempts to detect them, but we believe this is important future work.

This experimental demonstration is a proof of concept, showing how the behavior of some MIs, representative of those

observed to date, can be detected using assertion checkers that are based on behavioral requirements from an architectural specification. Because the experiment is a proof of concept for a novel method, rather than a quantitative comparison between methods, the total number of MIs and checkers is not of central importance, and adding more of either would not add value to this demonstration.

First, we implemented an open-source general-purpose processor model, whose design code and supporting tools are freely available, and created several “typical” MIs targeting it, based on examples in the literature. Next, we used the text of the processor’s architectural specification to generate a set of representative security assertions, describing various prohibited behaviors, in PSL, knowing *a priori* that some of the prohibited behaviors would be expressed by the MIs. We converted the assertions into assertion checkers, using our tool, and installed the checkers in the design. Finally, we modified the processor testbench and firmware to run, both with and without the MI triggers, and observed the checkers, verifying that they do detect the occurrence of the prohibited behaviors in question, when those behaviors occur.

We verified the behavior of the synthesizable checkers against commercial software-based assertion-checkers, using the same PSL formulas, by observing that the synthesizable checkers (installed in the design) and the simulation-only “soft assertions” agree (*hold* or *fail* together), at all clock cycles.

Although we already knew that some of the MIs we created would violate corresponding behavioral restrictions from the architecture, we used the experiment to determine the following:

- Can the behavioral restrictions be correctly translated into PSL assertions?
- Can the assertions be correctly<sup>2</sup> converted into checkers? Can the conversion be done efficiently, in terms of time and space required?
- Do the checkers correctly identify the offending behaviors, with no false positives and no false negatives?

We were able to answer all these questions affirmatively in our demonstration.

### A. OpenRISC Design

Because the HDL for commercial processors is protected intellectual property, we chose an open-source processor design for our demonstration. OpenRISC is an advanced open-source processor architecture, supported by many contributors, and hosted by the OpenCores organization [18]. The current CPU design is called the or1200. It has its own full MIPS-style basic 32-bit instruction set. OpenRISC processor designs have a pipelined execution unit, exception-handling units, data and instruction caches, memory-management units, a Wishbone bus, a debug unit, and support for peripherals via the bus. There are several OpenRISC implementations; the one we used for these experiments is called MINSOC, for “minimal system on chip,” designed by Fajardo [19]. MINSOC has an or1200 CPU plus on-chip memory, and includes Ethernet and UART units for input and output.

<sup>2</sup>In terms of the PSL formal semantics

## B. Malicious Inclusions

We designed two MIs for these experiments.

MI #1 allows a software process running in the processor’s User Mode to escalate its privilege level to Supervisor Mode, and is similar to other demonstrated MIs [2]. Once a process is running in Supervisor Mode, any number of software attacks are possible, as shown in the combined hardware-software attacks of King, et al. [2]. The MI on-trigger and off-trigger are unique opcode/data combinations that are extremely unlikely to occur unintentionally.

MI #2 is designed to be able to leak data from memory out the UART port. Upon receipt of the input trigger (“Get Data”) plus a memory location, the MI copies a sequence of bytes from the memory location, bypassing the normal memory access mechanism, and sends the data back out the port. No software is involved in this attack.

## C. Requirements, Assertions, and Checkers

In order to infer the processor’s security requirements, or behavioral restrictions, we reviewed the OpenRISC and MINSOC manuals for statements that dictated security-relevant behaviors. Though the manuals, like the processor design itself, are evolving, we identified around a dozen requirements that could be implemented as assertions. In some cases, the behavior of a certain signal or component may be only partially specified, i.e., it is required to behave in a certain way under certain conditions, but its behavior is otherwise not dictated. From a security standpoint, it is desirable that behaviors be fully specified - that is, a given behavior should be either *permitted* or *prohibited*, but not *neither, both, or unspecified*.

We implemented the ten requirements listed below, derived from the OpenRISC architectural manuals, using PSL assertions. Note there can be a one-to-many relationship between a requirement and the assertions needed to implement it, because often many signals and units, across varying components in a design, may collectively carry out a given function. For these examples, we use one or two assertions per requirement. A hardware behavior in the processor is *permitted* if it meets the conditions listed below, and *prohibited* otherwise.

- 1) *Group 0 special registers may only be modified if the CPU is already in supervisor mode.*
- 2) *Supervisor mode is only entered from User Mode on reset startup, or exception entry.*
- 3) *Exception handling is only entered if one of the identified exception mechanisms is activated.*
- 4) *Custom instruction opcodes are permitted, but must be declared in the implementation; unspecified custom instructions are not allowed.*
- 5) *Instructions in the interrupt servicing area of memory must only be accessed during exception processing, unless the processor is in supervisor mode, or in a reset.*
- 6) *A page fault must be generated if the MMU detects an access control violation for reads and writes.*
- 7) *The UART output signals may only change if a write has been commanded from the CPU.*

- 8) *A data change to the Ethernet output at the pad should only occur if a transmit has been commanded, or during Ethernet or CPU initialization.*
- 9) *The Debug Unit’s Value and Control registers are only accessible from Supervisor Mode.*

The behavioral requirements derive from the *architectural specification*, not the implementation, so the assertion-checkers operate independently of the design units they monitor.

## D. Simulation

To create our simulation and testbench, we implemented the MINSOC system-on-chip on Mentor Graphics’ QuestaSim simulator. We used the OpenRISC toolchain to cross-compile “firmware” to run on MINSOC, including test programs that did and did not include triggers for the MIs. We modified the MINSOC simulator testbench to send Ethernet and UART input data to MINSOC from the “outside world.” The firmware includes a bootloader that sets up the memory space, execution stack, and interrupts, then hands control to the main program.

We used psl2hdl to synthesize the ten assertions above into Verilog assertion-checkers, then added the Verilog modules to the design. We ran the simulation testbench with and without the MI triggers. Without the MI triggers, the testbench operates normally, and no assertions report failures. With the MI triggers, both MIs are activated. MI #1 elevates the processor from User Mode to Supervisor Mode directly, then returns it to User Mode. MI #2 reads the trigger’s memory location, surreptitiously copies eight bytes from memory starting at that location, and sends the data out the UART.

Results summary: As expected, the assertion-checker for Requirement #2 reports a failure when MI #1 operates, and the assertion-checker for Requirement #7 reports a failure when MI #2 operates. Otherwise, all assertions hold at all times; see Tables II for full results.

Table II  
OPENRISC TESTBENCH: ASSERTION STATUS, WITH MI TRIGGERS INACTIVE (LEFT) AND ACTIVE (RIGHT). NOTE THE CORRESPONDENCE BETWEEN THE SIMULATOR-ONLY SOFT ASSERTIONS AND THE SYNTHESIZABLE CHECKERS (HOLD AND FAIL AGREEMENT).

Rqmnt.	MI Triggers Inactive		MI Triggers Active	
	Soft Assertion	Checker	Soft Assertion	Checker
1	Hold	Hold	Hold	Hold
2	Hold	Hold	<b>*Fail*</b>	<b>*Fail*</b>
3	Hold	Hold	Hold	Hold
4	Hold	Hold	Hold	Hold
5	Hold	Hold	Hold	Hold
6	Hold	Hold	Hold	Hold
7	Hold	Hold	<b>*Fail*</b>	<b>*Fail*</b>
8	Hold	Hold	Hold	Hold
9	Hold	Hold	Hold	Hold

## E. Adding Coverage

One important limitation of trying to detect MIs using security checkers is the difficulty in triggering the MIs, if they employ rare-event triggers. Because of this challenge, security checkers may be best employed in conjunction with other techniques. For example, in normal verification, engineers run validation tests and collect “coverage” data, which might include what percentage of the HDL assignment statements was

executed. Engineers strive for 100% coverage, and compliance with all assertions, as reasonable assurance of a design’s proper implementation [10].

A circuit testbench is a software entity that is used to drive the inputs of a hardware design in simulation, so that the proper function of the circuit under test can be evaluated. Circuits that are not exercised by the testbench are flagged as suspicious. The rationale behind this approach is that a well-designed testbench should exercise all, or nearly all, portions of a design; sections not exercised could be the result of 1) an incomplete testbench, 2) an extraneous piece of the design, or 3) rare-event-triggered malicious circuitry. Our strategy is as follows:

- Design a thorough testbench, with the goal of achieving 100% coverage.
- Implement the assertion checkers.
- If we can reach 100% coverage (all forms - statement coverage, branch coverage, FSM coverage, etc.) in all design units and all of the assertion checkers hold, then the design does not violate the security policy, as expressed by the checkers (though the design could still violate a policy requirement that’s omitted from the checkers).
- If we cannot achieve 100% coverage but the checkers hold, at least the “covered” portion of our design does not violate the security policy, as expressed by the checkers, and we can focus on the “uncovered” portions for further (manual) analysis.

By using this combination of techniques, we can detect some MIs when they do activate, and constrain our search to a small portion of the design when they do not. To illustrate this point, Table III shows the decrease in statement coverage, in some of the “infected” design units, when the MI triggers are turned off. The larger the fraction of the design unit taken up by the MI, the larger the decrease when the MI is inactive.

Table III  
TESTBENCH STATEMENT COVERAGE IN SELECTED UNITS, WITH AND WITHOUT AN ACTIVE MI TRIGGER

Unit	MI Trigger Active	MI Trigger Inactive
uart_top.v	100%	50%
eth_rxethmac.v	95%	73%
or1200_if.v	100%	95%

Consider the following example. Suppose a Verilog design has a few lines of malicious code in it (which would be obfuscated in a real instance). Whenever the MI trigger is active, the circuits represented by this section of Verilog code will perform their function. Fortunately, these particular malicious behaviors are covered by some assertion-checkers. During a simulation run, executing these Verilog instructions corresponds to being “covered” during the run, such as by statement coverage (though we could also employ branch coverage and other forms). QuestaSim uses a checkmark (✓) to indicate that a statement has been covered, and an X to indicate that it has not.

In the first case, the simulation testbench has been run, but the testbench did not manage to trigger the malicious circuitry. The designer has not seen a security-checker violation, but

notices that 100% statement coverage has not been achieved, since some (X) marks remain:

```

✓ always @(posedge clk)
✓ begin
✓   if (secret_trigger)
X   begin
X     do_evil_signal <= 1'b1;
X     open_backdoor <= 1'b1;
X   end
✓ end

```

evil\_checker:  (holds)  
backdoor\_checker:  (holds)

In the second case, the simulation testbench has been run (perhaps for longer, or with a better mix of random input vectors), and this time the testbench did trigger the malicious circuitry. The designer has improved the statement coverage, possibly nearer to 100% on this module, but now the security checkers fail:

```

✓ always @(posedge clk)
✓ begin
✓   if (secret_trigger)
✓   begin
✓     do_evil_signal <= 1'b1;
✓     open_backdoor <= 1'b1;
✓   end
✓ end

```

evil\_checker:  (fails)  
backdoor\_checker:  (fails)

By combining security checkers and statement coverage in this manner, we can increase our assurance that the design obeys the specified security requirements, and at the same time focus our search for MIs on smaller portions of the design, rather than analyze every line of code.

## IV. ANALYSIS AND CONCLUSIONS

### A. Overhead

To get a rough idea of the performance impact of adding the checkers in this instance, we synthesized the MINSOC design using the Xilinx ISE for a Virtex-5 target FPGA. With just these ten assertion checkers added, the maximum speed was not affected (79 MHz either way), and the number of logic gates used increased .03%. Checkers derived from automata generally use little space in synthesis because each state of an automaton can be modeled using a single flip-flop, and the edges use a small number of AND-gates and OR-gates. Some of the automata in our other test cases have grown as large as 20-30 states, but all the automata used to instrument the MINSOC security assertions needed five states or fewer. During the phase in which the automata are composed with one another, our algorithm continually performs boolean simplification and automata trimming, eliminating states and transitions that are extraneous.

Separate from this investigation, we synthesized a set of 65 benchmark PSL assertions, pulled from various checker-generator publications, and determined the average number of flip-flops and logic gates a checker requires. We also analyzed the MIPS architecture [20], as a commercial example, to get an idea of roughly how many behavioral requirements would need to be modeled in a processor. Combining data from these two studies leads us to conclude (see [8] for details) that the necessary power and area overhead added by a complete set of checkers will normally range from approximately 1%-5% of a processor's total, depending on the number of requirements.

### B. Soundness and Completeness

It is important to be sure the synthesized assertion-checkers behave correctly, i.e., that they properly hold or fail on a given sequence, according to the defined semantics of the PSL formula [12]. Checker generation is a three-step process: the PSL formulas are rewritten into simplified form, automata are generated from the simplified formulas, and the automata are converted to synthesizable HDL checkers.

Morin-Allory, Boulé, Borrione, and Zilic used the PVS theorem prover to prove the consistency of the rewrite rules, with respect to the formal PSL semantics [21]. We have separately performed a structured verification of the soundness and completeness of both the automata construction and the generation of HDL modules from the automata, to establish end to end correctness of the method [22]; the analysis is omitted here due to space limitations.

As a cross-check, for this research we also implemented all the PSL assertions natively in QuestaSim; on each simulation run, we confirmed that the “soft” assertions in QuestaSim behave the same as our synthesized assertions (Table II).

### C. Strengths and Limitations

We assess the following as strengths of the method:

- *Persistence*. One advantage of evaluating security dynamically, rather than statically, is that static analysis of a high-level design does not account for malicious changes made afterward, to the low-level design or the physical system. Dynamic security evaluation, can detect MIs inserted *after* the high-level design phase (though a sophisticated attacker could emplace an MI *and* bypass the monitoring system at the same time; but the monitors make his task more difficult).
- *Early Frame of Reference*. Assertion-based security checkers do not rely on a trusted “golden sample” for reference; they appeal instead to requirements stated in design documents like the architectural specification. This mitigates the main problem of equivalence-checking: the trusted reference may itself be compromised.
- *Detection of Small MIs*. Physical analysis methods are limited in their ability to detect changes in a processor whose size is a small fraction (around .01%) of the overall processor size [23]. Design analysis methods, like the use of security checkers, can detect even very small changes to a design.

- *Compatibility*. Runtime security checkers can be used in concert with a variety of other techniques. For example, after designing security checkers for a processor, the processor design netlists can be obfuscated, to deter subsequent tampering, without affecting operation of the monitor units. As described above, design analysis methods that identify rarely-used or unused circuits can be applied together with security checkers in a complementary fashion.

And the following as limitations:

- *Level of Effort*. Creating security checkers for an entire design is a great deal of work to require of a processor designer, and adds to the normally significant effort of ordinary functional verification, which is closely related. The methodology also requires architectural designers to be far more complete in terms of describing permitted and prohibited behaviors, compared to the level of detail normally observed today.
- *Completeness*. Although most behavioral requirements will be dynamically enforceable, some may not be. Also, the implementation requires that every behavioral restriction in the specification be checked against every hardware module in the design related to that function; a complete implementation will often require multiple checker modules per behavioral restriction.
- *Modeling Memory*. Although the method works well for logic circuits, it does not appear well suited for storage circuits, such as cache or banks of RAM, since each memory unit would need its own checker. For these types of circuits, we believe it would be more appropriate to verify fidelity using other techniques, such as error-correcting circuits, encryption, hashing, etc.

### D. Related Work

Abramovici and Bradley proposed the use of “Security Monitors” within a fabricated design, but left open the details of how to construct the monitors [24], whereas we propose a specific methodology.

Love, Jin, and Makris recently proposed a design analysis method employing theorem-proving of security properties for hardware designs. They translate an HDL specification into an equivalent form in the “Coq” language and use automated tools to assist in the proofs. Their approach differs from ours in that it is static, rather than dynamic, not synthesizable, not in a native HDL, and has been tested on a small design unit rather than a general-purpose processor [25].

Zhang and Tehranipoor proposed a design analysis method that also uses a combination of code coverage techniques and assertions, but their assertions are not synthesizable and therefore only checked in software, and their method has been tested against smaller hardware design units, rather than general-purpose processors [26].

There are several proposed methods for inferring the presence of an MI in a processor, given a trusted reference sample, by detecting physical artifacts, for example small deviations in power or timing, by Jin and Makris [6], Chakraborty et al.

[27], and Rad, Tehranipoor, and Plusquellic [5]. These physical analysis methods are complementary to our efforts.

Several researchers have proposed MI-detection methods that concentrate on identifying infrequently excited circuits. “Trusted RTL” by Banga and Hsiao is one example [28]. The “Blue Chip” technique, by Hicks et al., identifies as suspicious those circuits not exercised by a testbench, then bypasses the suspicious circuits using interrupts and software emulation [7]. Our approach employs coverage similarly, but instead uses it in conjunction with dynamic checkers.

Waksman and Sethumadhavan describe how to counter the malicious insider threat by having the elements of a processor monitor data transactions among the other elements, in a form of mutual suspicion and checking [29]. Because MIs are often triggered by a rare event, and lie undetected otherwise, Waksman and Sethumadhavan have recently proposed a method for deterring MIs by preventing them from being triggered at all, using isolation and encryption techniques [30]. Both methods are independent of, but complementary to, our method.

### E. Future Work

For our proposed technique, and others, it would be worthwhile to perform an adversarial (i.e., double blind) experiment, in which MI designers compete against processor designers, so that the defenders have no knowledge of what circuits the attackers will target. The annual Embedded Systems Challenge has had a format like this, but for smaller designs [31].

As noted by Tehranipoor, a set of standardized metrics for MI detection methods would be useful [32]. Similarly, a public corpus of sample MIs targeting general-purpose processors would be a valuable resource.

## V. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful comments. This research was funded in part by National Science Foundation Grant CNS-0910734. The views presented in this paper are those of the authors and do not necessarily reflect the views of the United States Department of Defense.

## REFERENCES

- [1] Office of the Undersecretary of Defense for Acquisition, Technology, and Logistics, “Report of the Defense Science Board task force on high performance microchip supply,” Tech. Rep., February, 2005.
- [2] S. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, “Designing and implementing malicious hardware,” in *Proceedings of the 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats*. USENIX Association, 2008, pp. 1–8.
- [3] S. Adee, “The hunt for the kill switch,” *Spectrum, IEEE*, vol. 45, no. 5, pp. 34–39, May 2008.
- [4] X. Wang, M. Tehranipoor, and J. Plusquellic, “Detecting malicious inclusions in secure hardware: Challenges and solutions,” in *Hardware-Oriented Security and Trust, IEEE International Workshop on*, 2008, pp. 15–19.
- [5] R. Rad, J. Plusquellic, and M. Tehranipoor, “Sensitivity analysis to Hardware Trojans using power supply transient signals,” in *IEEE International Workshop on Hardware-Oriented Security and Trust*, June 2008, pp. 3–7.
- [6] Y. Jin and Y. Makris, “Hardware trojan detection using path delay fingerprint,” in *Hardware-Oriented Security and Trust, IEEE International Workshop on*, May 2008, pp. 51–57.
- [7] M. Hicks, M. Finnicum, S. King, M. Martin, and J. Smith, “Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically,” in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, May 2010, pp. 159–172.
- [8] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, “Security checkers: Detecting processor malicious inclusions at runtime,” in *IEEE International Symposium on Hardware-Oriented Security and Trust*, June 2011, pp. 34–39.
- [9] F. Schneider, “Enforceable security policies,” *ACM Transactions on Information System Security*, vol. 3, no. 1, pp. 30–50, 2000. [Online]. Available: <http://doi.acm.org/10.1145/353323.353382>; <http://doi.acm.org/10.1145/1168857.1168890>
- [10] S. Iman, *Step by Step Functional Verification with SystemVerilog and OVM*. San Francisco, CA: Hansen Brown Publishing Company, 2010.
- [11] M. Boule and Z. Zilic, *Generating Hardware Assertion Checkers*. Montreal, Canada: Springer, 2008.
- [12] IEEE, “Standard 1850-2010, for the Property Specification Language (PSL),” pp. 1–171, June 2010.
- [13] C. Eisner and D. Fisman, *A Practical Introduction to PSL*. New York, NY: Springer, 2006.
- [14] M. Boule and Z. Zilic, “Efficient automata-based assertion-checker synthesis of PSL properties,” in *Eleventh Annual IEEE International High-Level Design Validation and Test Workshop*. IEEE Computer Society, November 2006, pp. 69–76.
- [15] D. Baez. (2011, August) PLY (Python Lex-Yacc) Homepage. <http://www.dabeaz.com/ply/>.
- [16] R. Findenig, “Behavioral synthesis of PSL assertions,” Hagenburg, Austria, M.S. thesis. Upper Austrian University of Applied Sciences, 2007.
- [17] Graphviz Organization. (2011, August) Graphviz. <http://www.graphviz.org/>.
- [18] OpenCores Foundation. (2011, August) <http://opencores.org/>.
- [19] R. Fajardo, “Minimal OpenRISC system on chip user manual. OpenCores.org,” September 2010.
- [20] MIPS Technologies, Inc., “MIPS architecture for programmers, Vol. I–III,” 2010.
- [21] K. Morin-Allory, M. Boulé, D. Borrione, and Z. Zilic, “Validating assertion language rewrite rules and semantics with automated theorem provers,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 9, pp. 1436–1448, September 2010.
- [22] M. Bilzor, “Defining and enforcing hardware security requirements,” Ph.D. dissertation, Comp. Sci. Dept., U.S. Naval Postgrad. Sch., 2011.
- [23] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, “Trojan detection using IC fingerprinting,” in *Security and Privacy, IEEE Symposium on*, May 2007, pp. 296–310.
- [24] M. Abramovici and P. Bradley, “Integrated circuit security: New threats and solutions,” in *CSIIRW ’09: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research*. ACM, 2009, pp. 1–3. [Online]. Available: <http://doi.acm.org/10.1145/1558607.1558671>
- [25] E. Love, Y. Jin, and Y. Makris, “Enhancing security via provably trustworthy hardware intellectual property,” in *IEEE International Symposium on Hardware-Oriented Security and Trust*, June 2011, pp. 12–17.
- [26] X. Zhang and M. Tehranipoor, “Case study: Detecting Hardware Trojans in third-party digital IP cores,” in *IEEE International Symposium on Hardware-Oriented Security and Trust*, June 2011, pp. 67–70.
- [27] R. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, “MERO: A statistical approach for Hardware Trojan detection,” in *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES ’09. Springer-Verlag, 2009, pp. 396–410.
- [28] M. Banga and M. Hsiao, “Trusted RTL: Trojan detection methodology in pre-silicon designs,” in *IEEE International Symposium on Hardware-Oriented Security and Trust*, Anaheim, CA, June 2010, pp. 56–59.
- [29] A. Waksman and S. Sethumadhavan, “Tamper evident microprocessors,” in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, May 2010, pp. 173–188.
- [30] —, “Silencing hardware backdoors,” in *Security and Privacy, IEEE Symposium on*, May 2011.
- [31] Y. Jin, N. Kupp, and Y. Makris, “Experiences in hardware trojan design and implementation,” in *Hardware-Oriented Security and Trust, IEEE International Workshop on*, 2009, pp. 50–57.
- [32] M. Tehranipoor and F. Koushanfar, “A survey of hardware trojan taxonomy and detection,” *IEEE Design and Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.