Identifying Security Critical Properties for the Dynamic Verification of a Processor

Rui Zhang Natalie Stanley Christopher Griggs Andrew Chi Cynthia Sturton

The University of North Carolina at Chapel Hill

{rzhang, cgriggs, achi, csturton}@cs.unc.edu stanleyn@email.unc.edu

Abstract

We present a methodology for identifying security critical properties for use in the dynamic verification of a processor. Such verification has been shown to be an effective way to prevent exploits of vulnerabilities in the processor, given a meaningful set of security properties. We use known processor errata to establish an initial set of security-critical invariants of the processor. We then use machine learning to infer an additional set of invariants that are not tied to any particular, known vulnerability, yet are critical to security.

We build a tool chain implementing the approach and evaluate it for the open-source OR1200 RISC processor. We find that our tool can identify 19 (86.4%) of the 22 manually crafted security-critical properties from prior work and generates 3 new security properties not covered in prior work.

1. Introduction

Hardware companies conduct extensive testing and verification throughout the design phase, yet errata in the design persist to the final shipped product [2, 3]. And, just as is the case with software, bugs in the hardware can create vulnerabilities that are exploitable by malicious software [15]. Recent work has demonstrated the efficacy of using assertions built in to the hardware design to protect, post-deployment, against security vulnerabilities [10, 11, 22]. The assertions act as an execution monitor: each assertion is a proposition encoding a property that should always hold and at run-time the assertion monitors the hardware signals and state named in the property. If the property is ever violated the assertion fires, triggering an exception. In this way, the assertions act in concert with software to protect core, security-critical processor functionality. The question of what to assert – what are the properties that are critical to security – is an open one, and the problem addressed by this work.

The current state of the art is to develop the assertions manually by studying the processor's instruction set architecture (ISA), identifying properties of the ISA that are critical to the security of software running on the processor, and encoding those properties as assertions. The process requires human expertise and judgment, can be tedious and time consuming, and some properties that are important for security are obscure and unlikely to be identified. Furthermore, because the instruction manuals describing an ISA may be incomplete and ambiguous there are important properties which even the most thorough perusal of the ISA will be unable to uncover.

There is, however, a benefit to having a human in the loop. The line between a security property and a purely functional property is blurry. Some properties seem obviously critical to security. As an example, each of the above cited works ([10, 11, 22]) includes an assertion that the supervisor signal is set only in response to a small number of well defined events. Other properties, such as the one(s) violated by Intel's infamous FDIV bug [1, 7], feel safely characterized as purely functional. However, in general, making the distinction often comes down to a judgment call, one that weighs the cost of adding an additional assertion to the final design against the benefit of increased security provided by the particular assertion. Where the human expert chooses to draw the line between security and functional properties can change for different systems and at different points in time.

We present SCIFinder, a methodology and tool chain for semi-automatically generating a set of security-critical processor invariants that can be encoded as synthesizable assertions. Our approach is informed by three observations. First, detailed information about processor invariants may not exist in any specification documents; this information can only be learned by studying a running processor. Second, human expertise is still needed for, and well suited to, distinguishing security concerns from purely functional ones. And, third, properties that are critical to security tend to have some commonalities between them, for example, they concern state that is critical to security such as the supervisor signal.

ASPLOS '17, April 08 - 12, 2017, Xi'an, China

ISBN 978-1-4503-4465-4/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3037697.3037734

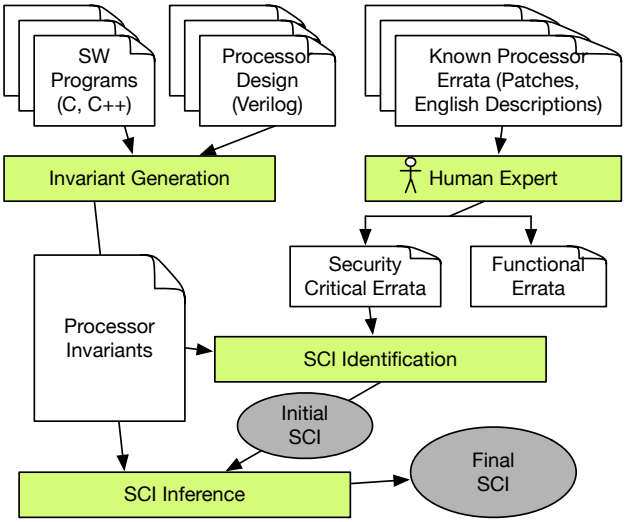

Figure 1: Workflow of SCIFinder.

Rather than try to cull security-critical properties from the ISA, we instead generate, automatically, a large set of processor invariants that describe all aspects of processor behavior and then categorize each invariant as critical to security or not. And, while we wish to use human judgment to guide this process, we do not want to burden the human with the task of combing through hundreds of thousands of invariants to perform the categorization. Therefore, we have developed two ways to algorithmically differentiate security-critical invariants from functional invariants: 1) Use published processor errata that can be shown to pose a threat to security to drive the categorization, and 2) Use statistical analysis techniques to classify the invariants.

The benefit of using published errata as a starting point is the potential to create assertions with high value. These assertions will at a minimum catch actual bugs that have had a deleterious effect on security. The approach has the potential to be stronger than that, however. If the assertions are well crafted, they capture not just the absence of a particular bug, but the presence of a desired security property. These assertions will detect any bug that violates the protected property, even if the bug itself is entirely different from the one that first inspired the assertion.

Still, the errata-based approach is limited to finding properties that have at one point been violated by a known security bug. In our second approach, we conjecture that once a human has identified points along the security–functionality boundary, machine learning techniques can be used to automate the classification of additional invariants. In this way new security-critical invariants can be identified.

SCIFinder has four phases (Figure 1):

1. Observe a processor executing a variety of programs to collect a set of likely processor invariants defined over software-visible processor state.
2. Given a list of known design errata, use human expertise and judgment to classify each erratum as either a functional bug or a potential security vulnerability.
3. Identify *security-critical invariants (SCI)* as those invariants violated by the security vulnerabilities.
4. Apply machine learning techniques to find additional SCI in the set of processor invariants.

In this work we focus on the development of a meaningful and comprehensive set of security-critical invariants. We present the following:

- Our two-pronged approach that uses known design errata plus machine learning techniques to identify processor invariants that are well suited to dynamic verification techniques and are critical to security.
- The implementation of our tool chain.
- An evaluation of our approach for the open-source OR1200 RISC processor, and a comparison of our semi-automated approach to the fully manual approach of prior work.

We find that our tool can identify 19 (86.4%) of the 22 manually crafted security-critical properties from prior work and generates 3 new security properties not covered in prior work. We test our generated assertions against 14 bugs from published AMD errata documents (bugs not used in the development of the assertions) and find the assertions stop 12 (86%) of these bug-based exploits.

2. Dynamic Verification for Security

Dynamic verification targeted for security is inspired by a long history of using assertion based verification (ABV) during hardware design, testing, and verification. In ABV, assertions are added to the hardware design, which is then simulated with random or selected inputs. Any assertions that fire during simulation point to a bug in the design [9, 17].

Unlike the assertions used as part of ABV, the security-critical assertions used in dynamic verification are kept in the design through synthesis and exist in the fabricated chip. So, while ABV assertions cannot detect bugs that are not triggered by test data, the security-critical assertions continue to monitor processor state and will detect, post deployment, attempts to exploit a vulnerability in the field. This form of dynamic verification has several benefits to recommend it. From a hardware-design point of view, the assertions are relatively simple statements with low overhead in terms of area and power. From a security point of view, the assertions can provide a powerful guarantee: any violation of the property will be detected, regardless of the possibly complex series of events that brought the processor to an insecure state.

What action is taken once the assertion fires depends on the system design. A simple design choice is to halt execution [10]; another option is to throw an exception to software. Hicks et al. found that software can often recover and move the processor past the buggy state to continue

making forward progress [22]. We leave this aspect of the system design as out of scope for this work.

3. Design

Our methodology has four phases: invariant generation; classification of known design errata as either security vulnerabilities or functional bugs; security-critical invariant (SCI) identification; and SCI inference. In the first phase, we collect a set of likely processor invariants. We use a modified version of Daikon, a dynamic invariant generation tool, and execute the processor design in simulation with a variety of software running on it. In keeping with prior art, we operate at the ISA level: we track software-visible state and consider execution of an instruction to be a single step of execution.

In the second phase we rely on human judgment to classify processor design errata as either security critical or not. We use as our source of errata the revision history for the design. This requires that revisions be documented with the reason for the change, and if it was a bug fix, a high-level description of the error. It is helpful if the revision documentation includes a test case that triggers the error.

The third phase relies on one of our key observations: security-critical errata are vulnerabilities precisely because they violate some underlying security property. We can use the errata to identify security-critical invariants – those that are violated when a security-critical erratum is triggered. SCI identified in this phase will protect against not just the particular vulnerability used to find it (and presumably that vulnerability has been patched in the latest version of the processor), but also against other, unknown bugs that violate the same invariant. In Section 5 we discuss how often this occurs within our test data.

In the fourth phase, we use machine learning techniques to identify additional invariants as security critical. We next discuss phases one, three, and four in detail. Details of our implementation of phase two are in Section 4.1.

3.1 Invariant Generation

We wish to collect meaningful processor invariants. We do this by generating a large number of processor execution traces covering as many processor states as possible, and then observing invariants within and across these traces. Some of the generated invariants are potentially security critical and will be identified as such in the following phases (see §3.3, §3.4).

3.1.1 Execution Traces

We obtain the processor execution traces by simulating the processor's register-transfer-level (RTL) design. During simulation we track architectural signals and selected register values of the RTL design at each instruction boundary. To provide as much breadth as possible, we run a variety of programs including SPEC benchmarks, a Linux boot, and scientific computations (see §5.1). Our execution traces must, at a minimum, cover all the instructions in the ISA, including system calls, bit-rotation operations, word-extension operations, and interrupts and exceptions.

3.1.2 Daikon

From the execution traces, we use Daikon, a dynamic invariant detection tool, to gather meaningful invariants [16]. Daikon has an instrumenter and an inference engine. The instrumenter records information about variable values as a program executes, and the inference engine reads the traces produced by the instrumenter to generate invariants.

Daikon is not specifically designed for hardware, and we adapted it to suit our needs:

1. Daikon is intended to learn software-level invariants: procedure pre- and post-conditions, class invariants, and data structure invariants. These are not directly applicable to processor execution traces; we extend Daikon to suit our hardware use case (see §3.1.3).
2. Patterns often seen in hardware design, such as bit-packing several flags into a single register, are unknown to Daikon. We develop new invariant patterns that capture such non-linear relationships between variables (see §3.1.4).
3. Certain processor design optimizations, such as delay slots, need to be carefully handled (see §3.1.5).
4. The invariants generated by Daikon contain redundancies. Our SCI will be enforced on processors dynamically and should be concise to avoid overhead. We introduce optimizations to remove redundancy (see §3.2).

3.1.3 Invariant Variables

Daikon produces invariants in the form of procedure pre- and post-conditions, as well as class and object invariants. The latter two are not applicable to our hardware setting, but the first two can be adjusted to suit our needs. We are interested in ISA-level properties that hold as the processor executes; by observing processor state before and after the execution of each instruction, we can use Daikon to develop a set of pre- and post-conditions for each instruction. The pre-conditions describe properties that always hold when a particular instruction executes, and the post-conditions describe properties that always hold at the conclusion of a particular instruction, provided the pre-conditions hold.

We modify Daikon's instrumenter to extract trace data from the execution logs produced by the simulation. It outputs variable values before and after each instruction is executed. The set of variables tracked should be inclusive enough for the inference engine to infer meaningful invariants including those critical to security. On the other hand, the variable set should be small enough to make invariant inferences computationally feasible.

We make the same design decision as prior work in dynamic processor verification. We include all the variables at the ISA level, that is, all registers and signals that are visible

to software: all general purpose registers (GPRs), all special purpose registers (SPRs), flags, data and address of the memory subsystem, target registers, and immediate values of the instruction. The ISA level represents a trade-off between complexity and completeness: the microarchitectural signals and registers that make up the processor implementation are abstracted away, reducing complexity. In exchange, we lose information that may be useful for constructing security properties. As an example, prior work found that an error in the processor's pipeline that modifies an instruction in flight would not be caught because the processor remains self-consistent at the ISA level. Extending our approach to capture microarchitectural information is the likely solution to this limitation. Our optimization strategies (§3.2) are a first step toward making such an extension feasible.

3.1.4 Invariant Patterns

Daikon invariants make comparisons between variables or between a linear combination of variables. We found this to be insufficient for capturing important properties at the hardware level. For example, a common pattern in hardware is for a 32-bit register to act as a record containing 32 (or fewer) independent bit flags. To address this, we made the Daikon instrumenter configurable. This allows users to create derived variables that can be used to define more complex invariants. For example, a derived variable that extracts bits from its parent variable can be used to generate a property indicating whether the flag that handles control flow is correctly set.

3.1.5 Processor Complexity

In many architectures, including the one in which we implement our tool, the processor always executes the instruction in the delay slot – the instruction directly after a control flow instruction (i.e., branch or jump). A naive observation would infer the invariant that the next program counter (NPC) after a control flow instruction is equal to the current program counter plus four (PC + 4), and while true, this does not capture the important property that control should move to the target of the branching instruction after executing the instruction in the delay slot. Similarly, the naive observation would be unable to infer an invariant about the NPC register for any other instruction. Normally a (non-branching) instruction obeys the invariant NPC = PC + 4, but if the instruction ever appears in a delay slot, its NPC would be the address of the branch or jump target.

To allow for the generation of meaningful invariants about control flow, we treat the control-flow instruction plus the one in the delay slot as a single entity. The OpenRISC architecture, the architecture we use in our implementation, has a single branch delay slot, so the branching instruction and the instruction in the delay slot is treated as one instruction. For those architectures with double branch delay slots (e.g. MIPS-X), the branching instruction and the pair of instructions following can be treated as one block.

$$
\begin{aligned}
EXPR &\doteq EXPR_1 \mid EXPR_2 \\
EXPR_1 &\doteq OPER\ OP_1\ OPER \\
EXPR_2 &\doteq OPER\ \texttt{in}\ \{imm, imm, \ldots\} \\
OPER &\doteq VAR \mid \texttt{orig}(VAR) \mid imm \\
OP_1 &\doteq = \mid \neq \mid < \mid \leq \mid > \mid \geq \\
VAR &\doteq \texttt{GPR} \mid \texttt{SPR} \mid flag \mid mem\_address \mid VAR \times imm \\
&\quad \mid \texttt{not}\ VAR \mid VAR\ \texttt{mod}\ imm \mid VAR\ OP_2\ VAR \\
OP_2 &\doteq \texttt{and} \mid \texttt{or} \mid + \mid -
\end{aligned}
$$

Figure 2: The grammar of invariant expressions. orig() indicates the value of a variable before the instruction executes; the default is the variable value after the instruction executes. imm refers to an immediate value. in indicates set inclusion. Boolean operators are all bitwise operators.

3.1.6 Structure of the Invariants

From the data generated during executions we use the Daikon generator to create invariants of the format

$$I \doteq risingEdge(INSN) \rightarrow EXPR,$$

where $risingEdge(INSN)$ represents the execution of an instruction, and $EXPR$ is an expression over the tracked variables. Figure 2 shows the grammar for expressions in our set of invariants.

As the execution of each instruction can take several cycles, we only consider the variables as they enter and leave the instruction. We designate the value of the variables before the instruction begins with the $orig()$ prefix, and any variable without the $orig()$ prefix indicates the value after the instruction has been completed.

To give an example, we show the invariant that describes the property that privilege should correctly de-escalate:

$$I \doteq risingEdge(\texttt{l.rfe}) \rightarrow \texttt{SR} = orig(\texttt{ESR0})$$

This invariant states that when returning from an exception (indicated by the l.rfe instruction), the status register (SR) should be correctly updated with the value it had before the processor entered the exception handler. ESR0 stores that value. The $orig(\texttt{ESR0})$ denotes the value of ESR0 before the l.rfe instruction is executed, while SR denotes the value of SR after the l.rfe instruction is executed.

We generate approximately 106,000 unique invariants which form a model describing normal processor behavior. Inherently the model we generate represents the current implementation of the processor; the correctness of our model is tied to the correctness of the implementation and design of the processor. Any errors or bugs in the specification and implementation will be reflected and remain undetected.

3.2 Optimization

We perform the following optimizations to put the invariants in a concise form.

3.2.1 Constant Propagation

Equality-to-constant invariants (e.g. $A = 0$) can be used to reduce the complexity of other invariants. Our constant propagation optimization is similar to the compiler optimization technique of substituting constant values at compile time [5, 6]. The propagation is performed iteratively so that any new equality-to-constant invariant can be used in subsequent substitutions.

We parse the invariants into expression trees, initialize a worklist with all the invariants, and construct a variable–value map. Then we iterate through the worklist, and for each invariant, we use the variable–value map to substitute constants for expressions where possible. For any new equality-to-constant invariant after substitution, we update the variable–value map and remove that invariant from the worklist. The process continues to iterate through the worklist until there are no new equality-to-constant invariants.

3.2.2 Deducible Removal

The deducible removal optimization pass removes the invariants that can be deduced from several other invariants. For example, $D < C$ is deductible from $A + B > D$ and $C > B + A$. Full deducible removal is equivalent to taking the transitive reduction of the binary relation; we remove invariants with transitive operators that can be derived from other invariants. Daikon invariants do not have complex expressions on both sides of an inequality, thus we do not perform deducible removal for cases similar to the following: $A + B > C + D$ is deducible from $A > B$ and $C > D$.

We first canonicalize invariants with transitive operators into the form of $lhs\ OP\ rhs$, where $OP \in \{>, \geq, ==\}$ ($<$ and $\leq$ will be converted accordingly), and lhs (rhs) is a sorted postfix string of the left (right) hand side of the expression. We build a directed acyclic graph (DAG) for all generated invariants for each OP. For each invariant $I \doteq lhs\ OP\ rhs$, we add the lhs and rhs as vertices in the DAG, and an edge directed from lhs to rhs. We then compute transitive reduction of the graph to get the minimum set of invariants with the same reachability relation.

3.2.3 Equivalence Removal

In this optimization pass we remove redundant invariants. We cluster invariants that are logically equivalent to each other in the same class and keep only one invariant from an equivalence class. For instance, the following invariants would be grouped into two equivalence classes and only two would be retained: $(A = B), (B = A)$; $(C + B * D > F), (F < C + D * B), (D * B + C > F)$, etc.

We determine invariant equivalence by putting every invariant into a canonical form, using the same form as used in the deducible removal pass.

3.3 Security-Critical Invariant Identification

Once we generate the set of invariants that describe normal processor behavior, our goal is to identify the subset of invariants that are crucial for security – the security critical invariants (*SCI*). One possible solution might be to use human expertise to develop a set of rules to apply. However, the rules may lack diversity: only the types of properties that a human has thought of will be represented, and prior work has shown that this approach can leave gaps in the resulting set of security properties [22]. In addition, the set of rules has to be small enough that the human can reasonably create it (i.e., there cannot be an individual rule for every generated invariant), but the rules themselves cannot be too general or they risk admitting too many invariants into the set of SCI.

For these reasons, we took an empirical approach to identifying SCI in the set of generated invariants. We leverage security errata that have existed in the processor design at some point in its development lifecycle. By definition, a program that triggers the bug must exhibit some unusual states that do not obey processor specifications. By checking which of our generated invariants are violated in the execution of a triggering program, we can approximately obtain the SCI. Because the errata are essentially programming bugs, they may occur anywhere in the design and potentially provide a more varied set of SCI than human-generated heuristics do. Because the identified SCI come directly from a security vulnerability, we know they are in fact critical to security.

To be specific, when we find a security bug from the published processor errata list or bug trackers (§4.1), we first implement the defect in an open source processor (in Verilog), creating a *buggy processor*. We then write a program that triggers the vulnerability, execute it on the buggy processor, and record its execution trace. Given the previously generated invariant set and the execution trace, our tool will automatically sort through the execution trace to see if *at any point* an invariant has been violated. Any violated invariants are then added to our set of candidate SCI.

Since the initial set of generated invariants may contain false positives, invariants identified as SCI in this step may not be true SCI. In order to remove these false SCI, we run the same trigger program on a correctly implemented processor (with the security defect removed) and perform the same steps of recording execution traces and checking for invariant violations. The set of violated invariants found in this phase are false positives, i.e. they are not true processor invariants, and can be eliminated from the final set of SCI.

One possible concern is that identified SCI are applicable only to one particular bug. In our experiments, we found that a single SCI can be identified from different bugs and it can stop multiple bugs (see §5.2). This means the SCI we extract from a particular bug are applicable to a class of bugs, a class defined by the invariant(s) violated.

3.4 Security-Critical Invariant Inference

Once we have identified a set of SCI using security-critical bugs, we apply machine learning techniques to infer which other invariants should be labeled security-critical.

The core component of the Inference step is a logistic regression model, which can be applied to classify invariants as security critical or non-security critical. We model the probability that an invariant is non-security critical as a function of its measured features. In particular, we adopt the penalized logistic regression model with elastic net penalty [34]. There are two reasons: 1) In this application the number of measured features is larger than the number of observations (invariants). Penalized logistic regression approaches have successfully extended traditional regression models for improved accuracy in such circumstances [34]. 2) This model excels in parameter interpretability [24]. As each feature included in the model incurs a cost or penalty, it can also be used to understand which of the features are critical to security.

Here, we specify the details of the regression model. We fit the model with the elastic net penalty using the glmnet [18] package in R.

As in the typical regression framework, we let $y_i \in \{\text{security-critical, non-security-critical}\}$ be the class label for invariant i. Since y_i is binary and hence a Bernoulli random variable, we model its probability, p_i, as follows.

$$\begin{aligned} p_i &= \text{Probability}(y_i = \text{non security critical}), \\ (1-p_i) &= \text{Probability}(y_i = \text{security critical}). \end{aligned} \tag{1}$$

For invariant i, we let $\mathbf{x}_i$ be its set of measured features. In our context, the features are all the ISA-level variables (§3.1.3) such as general purpose registers, flags, and memory addresses, and also operators such as $>, <, \neq$.

Then, we relate p_i to $\mathbf{x}_i$ as,

$$\log(\frac{p_i}{1-p_i}) = \mathbf{x}_i^T \boldsymbol{\beta} + \beta_0. \tag{2}$$

Here, $\boldsymbol{\beta}$ and β_0 are the vector of regression model coefficients and the intercept term, respectively, that are fitted with glmnet. The jth entry of $\boldsymbol{\beta}$ corresponds to the jth feature and explains that feature's contribution to the odds that invariant i is not security critical. β_0 is an intercept term giving the odds of being non security critical. When fitting the model, the objective is to learn the $\boldsymbol{\beta}$ and β_0 values that best describe the observed data.

We bootstrap this model using a small set of manually labeled invariants that contain both SCI and non-SCI. The constructed model can be used not only to predict whether a given invariant is likely an SCI but also to help hardware designers and security practitioners understand which of the features are critical to security based on the learned $\boldsymbol{\beta}$. For example, in our implementation only 24 of the 158 features have non-zero coefficients in the constructed models. These critical features include GPR0, PC, SF, ==, and IMM (see §5.3).

3.5 False Positives

False positives can occur in the final set of SCI in two ways. The first is that our tool generates an invariant that is not truly invariant. There are two potential sources for this type: 1) the Daikon tool itself; 2) inadequate test suites for invariant generation; and 3) the unintentional use of a buggy processor during the first stage. We minimize the first and second by tuning the parameters of Daikon to be conservative in finding invariants (see §5.1) and running many programs on our processor. (Increasing test coverage reduces the number of false positives.) The third source of false positive is a limitation of our tool. We rely on human experts to manually remove this kind of false positive from the final set of SCI.

The second type of false positive occurs when our tool classifies a non-security-critical invariant as security-critical. Reducing this type of false positives requires drawing a fine line to differentiate SCIs and non-SCIs, adding more labeled data, and refining machine learning models.

The issue of how invariants are integrated into the system – it is possible that false-positive invariants can be deployed to the processor – is beyond the paper's scope. Human experts can inspect the set of generated security-critical invariants to decide which are suitable for production use.

4. Implementation

Our tool is implemented mainly in Python. The exception is the SCI inference engine which is implemented in R and Matlab. As part of our evaluation we implement assertions enforcing the SCI on the OR1200 processor. This part of the work is implemented in Verilog.

4.1 Security-Critical Errata

We use potential security vulnerabilities to find security-critical invariants. We first collect bugs from the popular open source processors OR1200, LEON2, LEON3, OpenSPARC-T1, and OpenMSP430. Bugs are found from the processors' bugtracker and bugzilla sites, developers' mail archives, commits to the source repositories, comments in the source code, and published lists of errata. The bugs we collect are mainly in the core of the processor; bugs in peripheral devices such as UART, Debug Unit, and Ethernet are not included.

After collecting bugs, we manually select the bugs that may be classified as security critical: for each bug in the collection, we examine the patch and description to determine whether it is vulnerable to a security attack. In doing so we follow the same guidelines used by prior efforts in manually building SCI. Namely, we look for bugs that would allow an attacker to gain privileges to read or modify processor state that would not otherwise be allowed by the ISA or that would allow the attacker to subvert core functionality of the processor such as modifying the address in a load operation.

The total number of bugs we collected is 185, of those we deem 25 as security-critical. Of those 25, we successfully reproduced and modeled 17; 8 of them were not reproducible.

Bug No.	Synopsis	Source
b1	l.sys in delay slot will run into infinite loop	OR1200, Bugzilla #33
b2	l.macrc immediately after l.mac stalls the pipeline	OR1200, Bugtracker #1930
b3	l.extw instructions behave incorrectly	OR1200, Bugzilla #88
b4	Delay Slot Exception bit is not implemented in SR	OR1200, Bugzilla #85
b5	EPCR on range exception is incorrect	OR1200, Bugzilla #90
b6	Comparison wrong for unsigned inequality with different MSB	OR1200, Bugzilla #51
b7	Incorrect unsigned integer less-than compare	OR1200, Bugzilla #76
b8	Logical error in l.rori instruction	OR1200, Bugzilla #97
b9	EPCR on illegal instruction exception is incorrect	OR1200, Mail #01767
b10	GPR0 can be assigned	OR1200, Mail #00007
b11	Incorrect instruction fetched after an LSU stall	OR1200, Bugzilla #101
b12	l.mtspr instruction to some SPRs in supervisor mode treated as l.nop	OR1200, Bugzilla #95
b13	Call return address failure with large displacement	LEON2, Amtel-errata #2
b14	Byte and half-word write to SRAM failure when executing from SDRAM	LEON2, Amtel-errata #3
b15	Wrong PC stored during FPU exception trap	LEON2, Amtel-errata #4
b16	Sign/unsign extend of data alignment in LSU	OpenSPARC T1
b17	Overwrite of ldxa-data with subsequent st-data	OpenSPARC T1

Table 1: Security-critical bugs implemented and used for evaluation.

Table 1 shows the 17 security-critical processor bugs we use. The first 12 bugs are from OR1200, 3 bugs are from LEON2, and the last 2 are from OpenSPARC T1.

Bugs b1 and b2 may allow denial-of-service (DoS) attacks. In particular, bug b1 causes the processor to run in an infinite loop and bug b2 stalls the pipeline infinitely. Although the attacks violate liveness properties, we can identify security-critical safety properties at the root of the vulnerability. For example, the SCI we identified for b1 shows that the root cause of the vulnerability is that the PC is not correctly updated.

Bug b8 can be exploited to make the processor ignore an exception that it should handle. Attackers may leverage this to bypass some security checks. For example, failing to raise a bus error exception will potentially allow users to write into protected memory area.

Bugs b6, b7, or b13 leave the processor open to insecure control flow. Attacking bug b6 or b7 will cause the processor to incorrectly set the flag that decides whether branches should be taken. As a result, the processor may execute a sequence of instructions of the attacker's choosing. Bug b13 will incorrectly set the link register, which will cause the processor to return from a function call incorrectly and thus run a sequence of unexpected instructions.

Bug b11 can cause the processor to execute the wrong instruction. Even though the processor would execute the instruction correctly, the instruction itself in the pipeline has been contaminated because of subtle timing constraints. This allows the attackers to change or substitute instructions according to their needs.

Attacks on bug b12 can cause `l.mtspr` (Move to Special-Purpose Register Instruction) to act as a no-op when moving the content of a general-purpose register to some special-purpose registers. This bug causes the processor state to be incorrectly updated.

Bugs b4, b5, b9, and b15 deal with the contamination of exception-related special-purpose registers. This exposes the processor to security vulnerabilities because contaminating the registers that store the pre-exception processor state can potentially lead to privilege escalation.

Bugs b3, b10, b14, b16, and b17 are related to memory access. Bugs b3 and b10 can cause an incorrect address calculation or the wrong data to be loaded or stored. Bugs b14, b16, and b17 contaminate the data transferred between memory subsystems and registers. A potential attack might be to modify secret keys by contaminating the memory address or the data itself when loading or storing the data.

We reproduced these 17 bugs in the OR1200 processor, which is a 32-bit implementation of the OpenRISC 1000 architecture with Harvard microarchitecture, 5-stage integer pipeline, virtual memory support (MMU), and basic DSP capabilities [23]. Our processor implements the basic instruction set (i.e., none of the extension modules such as floating point). It is widely used in research projects and embedded computer environments. For each bug we also developed a triggering program written in a mixture of C and assembly that attacks the buggy processor and causes the violation of some security policies during execution.

4.2 Assertions

Our tool does not yet provide the automatic translation from SCI to hardware assertions enforcing those SCI. However, in our experience the process is straightforward and we give an example showing what is required for translation from an invariant to an RTL assertion. We leverage the industry standard Open Verification Library (OVL) for constructing assertions. All SCI were translated using one of four OVL

assertion templates: *always*, *edge*, *next*, *delta*. *always* is used when the expression is always true; *edge* is used when the expression is true at the point when the instruction is sampled; *next* is used when the expression is true some number of clock cycles after sampling the instruction; *delta* is used when a monitored signal's updates stay within a range.

Taking the invariant we described in 3.1.6 as an example,

$$I \doteq risingEdge(l.rfe) \rightarrow SR == orig(ESR0),$$

the corresponding assertion for this invariant is

$$A \doteq next(INSN = l.rfe, SR = ESR0_{PREV}, 1).$$

This means expression $SR = ESR0_{PREV}$ must be true one clock cycle after instruction l.rfe is sampled. Note that we need to store the previous cycle value of $ESR0$.

5. Evaluation

In this evaluation we show that 1) our tool effectively generates SCI from existing security-critical bugs; 2) the generated SCI stop both the existing security-critical bugs and new bugs; 3) meaningful SCI not tied to any known security-critical bugs can be found; and 4) the automatically generated SCI represent security properties written by experts.

5.1 Invariant Generation

Our tool's first step is to run a variety of programs on the processor to generate candidate invariants. We collected 26GB of trace data from 17 programs; more trace data results in more accurate invariant generation. We configured Daikon with a confidence limit of 0.99, reducing the risk of generating false-positive invariants that hold by chance in our trace data set. The filters search for invariants matching our invariant grammar in Figure 2.

We evaluate how the number of programs affect the set of invariants generated. We use the following programs: Linux boot, SPEC benchmarks (Parser, Mesa, Ammp, Mcf, Instru, Gzip, Crafty, Bzip, Quake, Twolf, Vpr), Basicmath, Pi Calculation, Bitcount, FFT, Helloworld. The execution traces cover all 56 instructions of the OpenRISC (basic instruction set) architecture. Figure 3 shows the result of this evaluation. We see that running additional programs may add invariants to the result set by exercising new features of the processor. It may also eliminate some invariants from the result set that cannot be justified by the new trace.

The overall trend of Figure 3 indicates that as the number of programs increases, the set of unique invariants that we generate becomes stable. After adding the *twolf* benchmark, no new invariants are generated or removed. From this trend, we extrapolate that if we run enough (finite) programs on the processor, we will reach a stable set of invariants that can roughly model the behavior of a processor.

After the initial set of invariants is generated, it is optimized. Table 2 shows the effectiveness of different optimization passes in reducing redundant and lengthy raw invariants.

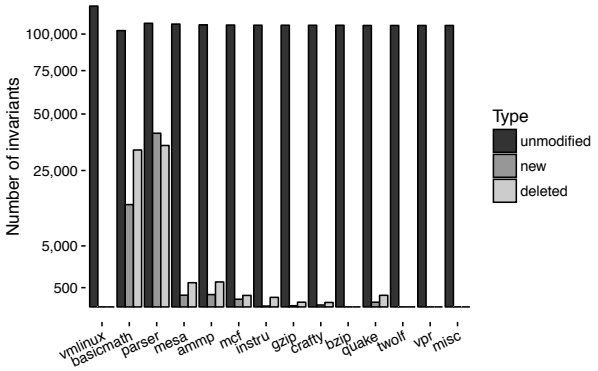

Figure 3: Unique invariants generated from executing programs. The X-axis is aggregative, e.g., `basicmath` means invariants generated from running both `vmlinux` and `basicmath`.

	Raw	after CP	after DR	after ER
Invariants	106,174	106,174	90,955	88,301
Variables	210,013	171,858	170,517	167,863

Table 2: Effect of invariant optimizations (§3.2) in reducing the total number of invariants and variables in all invariants. CP is constant propagation; DR is deducible removal; ER is equivalence removal.

The optimizations in combination achieve 17% reduction in terms of the number of invariants and 20% reduction in terms of the number of total variables in all invariants.

5.2 SCI Identification

The second step for our tool is SCI identification. Given a set of optimized invariants, a buggy processor and a triggering program, our tool identifies the affected SCI from the invariant set. Table 3 shows the number of identified SCI for each of the 17 security-critical bugs we implemented.

In total, our tool identifies SCI for 16 (94%) of the 17 bugs. Interestingly, although bug b1 and b5 are two different bugs, our tool identified the same SCI. This shows one advantage of our tool: the SCI we extract from a particular security bug are not just applicable to that bug, but rather potentially to a class of bugs. The only bug for which our tool fails to identify any SCI is bug b2. The reason is that no ISA-level invariants are violated by this bug. The bug is in the pipeline and all software-visible signals remain self-consistent. Identifying SCI for this bug would require adding microarchitectural level variables to Daikon's instrumenter and generating microarchitectural level invariants.

Table 3 also shows more than one SCI identified per bug in some cases. This occurs for one of three reasons. The simplest is that the bug violates more than one security property. A second reason is that violating a single property may have multiple consequences. For example, in our implemen-

Bug No.	True SCI	FP	Detected
b1	2	22	✓
b2	0	N/A	×
b3	1	8	✓
b4	2	2	✓
b5	5	28	✓
b6	1	5	✓
b7	1	1	✓
b8	3	0	✓
b9	4	0	✓
b10	32	0	✓
b11	1	0	✓
b12	1	4	✓
b13	2	0	✓
b14	1	0	✓
b15	1	25	✓
b16	1	0	✓
b17	3	2	✓

Table 3: SCI identified from the 17 security-critical bugs we reproduced (see Table 1). Detected means enforcing the SCI as assertions on the processor can detect the buggy behavior dynamically.

tation the syscall handler is always at address `0xC00`. Bug b8 violates this property and, therefore, the two invariants `l.sys` $\rightarrow$ `PC` $=$ `0xC00` and `l.sys` $\rightarrow$ `NPC` $=$ `0xC04`, where `l.sys` is the syscall instruction, `PC` is the program counter, and `NPC` is the next program counter. A third reason is that a violation may persist for multiple steps and our SCI are defined per instruction. For example, bug b10 violates the property `GPR0` $= 0$. The bug manifests in the add instruction and violates the invariant `l.add` $\rightarrow$ `GPR0` $= 0$. And, as the register is not restored to a valid state subsequent instructions violate analogous invariants, such as `l.nop` $\rightarrow$ `GPR0` $= 0$.

The set of identified SCI may include false positives. We manually validated the identified SCI and found 7 of the bugs (43.8%) resulted in 0 false positives, while 6 of the bugs (37.5%) resulted in fewer than 10 false positives (Table 3).

In practice, the false positives in the identified SCI can be easily spotted (e.g., an SPR must equal 0). We envision the usage scenario of our tool is that after it identifies SCI, experts would validate them before putting into a processor.

To further validate that our automatically identified SCI are useful, we enforce them as assertions in a SPECS-like system. The result shows that all the 16 security-critical bugs from which we identified SCI are detected dynamically, meaning the SCI are effective.

5.3 SCI Inference

In Section 5.2 we show that the SCI we build from the Identification step can effectively detect security-critical bugs and some identified SCI can detect multiple different bugs. In this section, we show that our tool can identify useful SCI not tied to any particular previously known bug. We use an elastic net logistic regression model to infer new SCI from existing SCI.

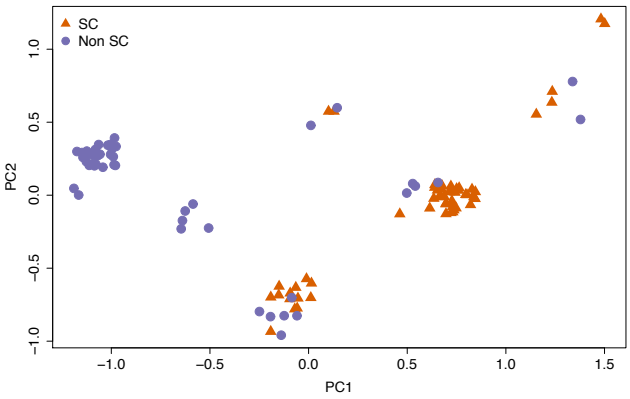

Figure 4: PCA using selected features. From the learned elastic net logistic regression model, 24 of the original set of 158 features had non-zero coefficients. PCA was performed using the 24 selected features on 102 SCI/non SCI. The plot shows the projection of these invariants in 2 dimensions.

We start with our 88,301 invariants, each with 158 features, i.e., in our model from Section 3.4, $N = 88,301$, $P = 158$. Our model is supervised, and we leverage the results from the Identification step to provide labels to train the model. In particular, we have 54 verified SCI (*unique* SCI in Table 3). We label the *unique* false positives from the Identification step as non-SCI, a total of 48 invariants.

Of these 102 labeled invariants, we used 70% of the data as training data and performed the optimization of $\boldsymbol{\beta}$ and β_0 using the glmnet [18] package in R. We took $\alpha = 0.5$ and used 3-fold cross validation in the training set to choose an appropriate λ. Doing so, resulted in $\lambda = .08$. When we tested the model on the test set, we observed 90% accuracy, validating the quality of the fitted model.

In the constructed model, there were 24 non-zero coefficients from the original set of 158 features (see Table 4). To evaluate how these 24 features can be used to partition invariants in high-dimensional feature space, we performed principle component analysis (PCA) on the 102 labeled invariants according to this limited set of 24 selected features. Figure 4 shows the projection of these invariants in 2-dimensional space. As expected, using this set of features, invariants cluster adequately according to class label. This supports the model's selection of features as robust candidates for distinguishing SCI from non SCI.

We use the constructed model to further predict the entire set of 88,199 (88,301−102) *unlabeled* invariants. Table 5 shows the results. The model recommends 3,146 out of the 88,199 invariants as SCI. In the Identification step, we used the triggering programs to validate an identified SCI. In this Inference step, we do not have ground truth for the 88,199 invariants, but we manually examined the 3,146 recommended SCI and spotted 852 clear false positives.

Weight	Features			
Positive	GPR6 IM orig(IM)	OP_B MEM_{BUS} $<$	ROR orig(OP_A) $\neq$	DIV orig(SPR) $+$
Negative	GPR0 IDPC orig(NNPC)	PC REG_B CONST	SF orig(GPR0) ==	WBPC orig(NPC) >=

Table 4: 24 identified features with non-zero coefficients. Features with negative weights are associated with SCI. Features with positive weights are associated with non-SCI.

Invariants	Inferred SCI	FP	Security Properties
88,199	3,146	852	33

Table 5: SCI inference results

These inferred SCI can be concisely described as 33 security properties that can be added, in the form of assertions, to a processor. In Section 5.4, we show that some of the inferred SCI represent security properties that are not covered by the SCI found in the Identification step, demonstrating the advantage of SCI inference.

5.4 Representing Manually Written Security Properties

To evaluate the efficacy of our tool, we test whether it finds SCI, either from Identification or Inference, that represent the manually written security properties of the two state-of-the-art works: SPECS [22] and Security-Checker [11].

Table 6 shows the result. Of the 27 security critical properties from these two papers, 3 (p25, p26, p27) are security bugs outside of processor cores. These are not the target of this paper. For the remaining 24, 2 of them (p18, p24) need microarchitectural states and thus our tool cannot generate these two invariants. Thus, we mainly focus on whether our tool can identify or infer the remaining 22 security properties using the 17 security-critical bugs we reproduced.

From the Identification step, 11 (50%) of the 22 security properties are identified from 12 out of 17 bugs. There are three interesting findings. The first is that a single security property can be identified from different bugs and the identified SCI are different. For example, for bugs b4, b9, and b15, the identified SCI are different although they belong to the same security property (p3). The second is that different security properties can be identified from the same bug, e.g. p13 and p14 can be identified from b5. Finally, a single SCI can concisely represent multiple manually written security properties, e.g. p17, p21 and p23. The SCI for these properties is $risingEdge(\texttt{1.sys}) \rightarrow \texttt{PC} = \texttt{0xC00}$.

Adding the Inference step, 8 (36%) additional security properties are found. Two (p10 and p22) are not found be-

No.	Security Property Description	Class	From Ident.	From Infer.
	Properties from SPECS [22]			
p1	Execution privilege matches page privilege	XR		✓
p2	SPR equals GPR in register move instructions	RU	b12	
p3	Updates to exception registers make sense	XR	b4 b9 b15	
p4	Destination matches the target	CR		✓
p5	Memory value in equals register value out	MA	b14	
p6	Register value in equals memory value out	MA	b16 b17	
p7	Memory address equals effective address	MA		✓
p8	Privilege escalates correctly	XR		✓
p9	Privilege deescalates correctly	XR		✓
p10	Jumps update the PC correctly	CF		▲
p11	Jumps update the LR correctly	CF	b13	
p12	Instruction is in a valid format	IE	b11	
p13	Continuous Control Flow	CF	b5	
p14	Exception return updates state correctly	XR	b1 b5	
p15	Reg. change implies that it is the instruction target	CR		✓
p16	SR is not written to a GPR in user mode	RU		
p17	Interrupt implies handled	XR	b8	
p18	Instr unchanged in pipeline	IE		★
	Properties from Security-Checker [11]			
p19	SPR modified only in supervisor mode	RU		✓
p20	Enter supervisor mode is on reset or exception	XR		✓
p21	Exception handling implies exception mechanism activated	XR	b8	
p22	Unspecified custom instructions are not allowed	IE		▲
p23	Exception handler accessed only during exception, in supvr mode, or on reset	XR	b8	
p24	Page fault generated if MMU detects an access control violation	MA		★
p25	UART output changes on a write command from CPU			■
p26	Only transmit cmd or initialization change Ethernet data output			■
p27	Debug Unit's value and ctrl regs only accessible from supvr mode			■

Table 6: Evaluation against security properties from prior work. For each property we indicate whether it was found in the identification (From Ident) or the inference (From Infer) step. The bug numbers correspond to Table 1. ✓means the property is found. If the property is not found it may be because it is not generated from Daikon (▲), it needs micro-architectural state (★), or it relates to HW outside the processor core (■).

No.	Security Property Description	Class	From Ident.	From Infer.
p28	Flags that influence control flow should be set correctly	CF	b6 b7	
p29	Calculation of memory address or memory data is correct	MA	b3 b10	
p30	Link address is not modified during function call execution	CF		✓

Table 7: New security properties generated by our tool that are not covered in prior work.

cause they do not exist in the invariant set generated with Daikon, and one (p16) is not identified as security critical although it does exists in the set of generated invariants.

Property p10 is missing because Daikon does not capture effective addresses (the immediate value shifted left two bits, sign-extended to program counter width, and then added to the address of the jump/branch instruction [23]). By adding the effective address as a derived variable to Daikon, we can generate this invariant. Property p22 is missing because it concerns custom instructions, which are part of the extended instruction set that we did not implement. (Recall, we implement the basic instruction set in our evaluation.)

Property p16 is not found by our tool, although the associated invariant does exist in our generated set of invariants. The invariant is $risingEdge(\texttt{l.add}) \rightarrow \texttt{SR} \neq \texttt{OP}_{\texttt{DEST}}$. It is neither violated by any of our implemented bugs, nor is it labeled as security critical by our logistic regression model. The latter is because in our model the $\neq$ operator is a feature with high positive weights, meaning invariants with that operator are likely to be classified as non-security-critical.

Our tool generates 3 new security properties not found by either SPECS or Security-Checker (Table 7). Two properties (p28, p29) are identified from bugs during the Identification phase, and one (p30) is from the Inference phase.

The property (p28) identified from bugs b6 and b7 is an example of using a derived variable, in this case one that describes the behavior of correctly setting the control flow flag. The property (p29) identified from bugs b3 and b10 is related to calculation. We note that SCIFinder is able to differentiate between calculations often used for memory addresses and others, and labels only the former as security critical. For example, the property GPR0 = 0 is often leveraged during address calculation and SCIFinder identifies multiple SCI to enforce it. Whereas invariants related to rotate calculations are not identified as security critical.

The property found during the Inference step (p30) has to do with the link address. A link address gives the location of a function call instruction and is used to calculate where program execution should return after function completion [23]. The inferred SCI states that the link address should not be modified during function execution.

5.5 Classification of Security Properties

The SPECS project classified security-critical processor errata into five classes (invalid register update, execute incorrect instruction, memory access, incorrect results, and exception related) [22]. Inspired by this, we classified the security properties related to the processor core into six classes: five of them are similar to the SPECS classification and we add one new class that is related to control flow. The classification results are shown in Tables 6 and 7.

CF stands for control flow related properties; **XR** stands for exception related properties; **MA** represents properties related to memory access; **IE** stands for the class of security properties that guarantee the processor will execute the correct and specified instructions; **CR** represents the class of security properties about correctly updating results.

Classifying the properties yielded two observations. The first is that SCIFinder was effective at finding properties related to exceptions (XR). Of the 27 properties identified by prior work, 9 fall into the XR category (the largest category by far – CF and MA are the next largest with 5 properties each) and SCIFinder was able to find all 9. On the other hand, SCIFinder was least effective for properties related to instruction execution (IE). Of the three identified in prior work, SCIFinder found only one. The two missed properties, p18 and p22, required microarchitectural state and analysis of custom instructions, respectively. We caution that these are observations; the total number of properties is too small to draw conclusions. However, they do suggest areas where SCIFinder may shine, as well as opportunities for future research to strengthen the SCIFinder approach.

5.6 Detecting Unknown Bugs

The SCI have the potential to stop new bugs that have not been seen before. We cannot measure this directly, as new bugs would only be found if we happened to run software that triggered the bug (causing the SCI assertion to fire). Instead, we took a set of bugs that we had not used in our identification or inference phases, added them to the processor, and ran software that triggers the bugs to see whether our SCI would fire. For this experiment we use the 14 AMD errata from the SPECS project. The authors reproduced the errata in the OR1200 processor and made their code public. Our tool is able to detect 12 of the 14 bugs. (By way of comparison, SPECS was also able to detect 12 bugs.) Five of these were detected by the Identified SCI, while seven were detected by the Inferred SCI. This demonstrates that our automatic SCI are not just applicable to the 17 known bugs from which they were generated, but are also useful to detect unknown bugs.

To avoid selection bias we repeat the experiment, but this time we randomly pick 14 bugs from our set of 28 (both from design documents and from AMD errata lists, excluding the 3 that use microarchitectural state), for use in the Identification and Inference steps. We use the remaining

Step	Data	Size	Time hh:mm:ss
Invariant Generation	traces	26GB	11: 21 :00
Optimization	invariants	106,174	00: 00 :04
SCI Identification	invariants +bugs	88,301 +16	00: 44 :52
SCI Inference	invariants	88,301	<00: 00 :01

Table 8: Execution time. Except for traces, sizes are given as number of items, e.g., the inference phase reads in 88,301 invariants.

	Baseline	Initial SCI	Final SCI
Logic	10073 LUTs	1.6%	4.4%
Power	3.24 W	0.13%	0.31%
Delay	19.1 ns	0%	0%

Table 9: Hardware overhead. The baseline is the OR1200, Xilinx xupv5-lx110t-based System-on-Chip. Initial SCI are the 14 assertions from Identification step. Final SCI are the 33 assertions from both Identification and Inference steps.

14 bugs for testing. Of the test set, only bug b6 is not detected; the SCI for detecting b6 ($risingEdge(\texttt{l.sfleu}) \rightarrow (\texttt{OP}_\texttt{A} - \texttt{OP}_\texttt{B}) * (1 - 2 * \texttt{CF}) \geq 0$) is not found.

5.7 Performance

In this section, we evaluate the performance of our tool. The experiments are performed on a machine with an Intel Core i7 Processor (quad-core, 2.60GHz) and 8 GB of RAM. Table 8 shows the CPU time taken for each step of our tool. The whole process takes about 12 hours. The most expensive step is the Invariant Generation for 26 GB of trace data. In practice, a full Invariant Generation step is only performed once and all subsequent generation is incremental.

We also report on the manual effort needed to validate the SCI recommended by our tool. It took a graduate student roughly 5 hours to go through the entire list of 3,146 recommended SCI to identify false positives (10.5 invariants per minute, on average). Invariants pertaining to one instruction were carefully validated first, and those that were clearly non-invariant (as determined by the ISA) were classified as false positives. Invariants with the same expression as the false positives, but pertaining to other instructions, were then easily searched for and eliminated. Finally, invariants that belong to only one or a few instructions were validated.

Finally, table 9 shows the hardware overhead incurred by adding our assertions to the OR1200 design. The additional logic is less than 5% of the original design, incurs a power overhead of 0.3%, and adds no delay.

6. Related Work

Use of Security Critical Assertions Prior work has established the use of assertions post deployment to strengthen the security of hardware [4, 10, 11, 22]. Our work builds upon this literature and we evaluate our semi-automatically generated invariants against the manually crafted invariants of prior art.

Extracting Assertions from HW Designs The IODINE tool automatically extracts from designs ABV assertions such as one-hot encoding or mutual exclusion between signals [19]. More recent papers use data mining of simulation traces to extract assertions [12, 21]. These approaches focus on extracting assertions for functional verification and are not concerned with finding processor's security properties.

Approaches for Protecting Vulnerable HW Techniques for detecting and recovering from security-critical processor bugs fall into three categories: hardware-based [8, 27, 29, 30], software-based [20, 25, 32] and a hybrid of hardware and software [13, 14, 22]. Hardware-based solutions include adding redundancy to protect against random errors and checking processor state transitions against a known set of errata signatures. Software-based solutions include microcode patching and binary translation. The hybrid approaches can provide the best of both worlds: high coverage with low overhead; the use of assertions for dynamic verification post-deployment falls into this category.

Data Mining for Security Properties of SW Security properties in software have been found using human specified rules [31], by observing instances of deviant behavior [16, 26, 28], or by identifying instances of known bugs [33].

7. Conclusion

We have presented SCIFinder, a methodology and tool-chain for generating security-critical invariants (SCI). Given a list of known security-critical errata from a processor and the processor design we identify a set of SCI that can be used to dynamically verify the processor's security. Experiments show SCIFinder's practicality and effectiveness in generating meaningful SCI. It identifies effective SCI for 16 of 17 bugs from input errata plus 12 bugs from AMD errata lists. The final SCI set covers 86.4% of the manually crafted security properties and identifies 3 new properties not covered in prior work.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful questions and constructive suggestions for improvement. This research was supported by the National Science Foundation under grants CNS-1464209 and CNS-1651276, and the National Institutes of Health under grants T32CA201159. Any opinions, findings, conclusions, and recommendations expressed in this paper are solely those of the authors.

References

[1] Intel pentium processor statistical analysis of floating point flaw. *Intel White Paper*, July 2004.

[2] Revision Guide for AMD Family 16h Models 00h-0Fh Processors. *Product Revision*, 2013.

[3] Intel Core i7-600, i5-500, i5-400 and i3-300 Mobile Processor Series. *Specification Update*, 2014.

[4] M. Abramovici and P. Bradley. Integrated circuit security: New threats and solutions. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, CSIIRW '09, pages 55:1–55:3, New York, NY, USA, 2009. ACM.

[5] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[6] F. E. Allen. Program optimization. In *Annual Review in Automatic Programming, vol. 5*, pages 239–307, 1969.

[7] D. Athow. Pentium FDIV: The processor bug that shook the world. *techradar.pro*, October 2014.

[8] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 196–207, 1999.

[9] A. A. Bayazit and S. Malik. Complementary use of runtime validation and model checking. In *Proceedings of the 2005 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '05, pages 1052–1059, Washington, DC, USA, 2005. IEEE Computer Society.

[10] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin. Security checkers: Detecting processor malicious inclusions at runtime. In *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, pages 34–39, June 2011.

[11] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin. Evaluating security requirements in a general-purpose processor by combining assertion checkers with code coverage. In *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*, pages 49–54. IEEE, 2012.

[12] P.-H. Chang and L. C. Wang. Automatic assertion extraction via sequential data mining of simulation traces. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 607–612. IEEE, 2010.

[13] K. Constantinides and T. Austin. Using introspective software-based testing for post-silicon debug and repair. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 537–542, June 2010.

[14] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-based online detection of hardware defects mechanisms, architectural support, and evaluation. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 97–108, Dec 2007.

[15] T. de Raadt. Intel Core 2. OpenBSD-misc mailing list, June 2007. `http://marc.info/?l=openbsd-isc&m=118296441702631;`.

[16] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, Dec. 2007.

[17] H. Foster, A. Krolnik, and D. Lacey. *Assertion-Based Design*. Springer US, 2005.

[18] J. Friedman, T. Hastie, and R. Tibshirani. glmnet: Lasso and elastic-net regularized generalized linear models. *R package version*, 1, 2009.

[19] S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty. IODINE: A tool to automatically infer dynamic invariants for hardware designs. In *Proceedings of 42nd Design Automation Conference*. IEEE, 2005.

[20] L. C. Heller and M. S. Farrell. Millicode in an IBM zSeries processor. *IBM Journal of Research and Development*, 48 (3.4):425–434, May 2004.

[21] S. Hertz, D. Sheridan, and S. Vasudevan. Mining hardware assertions with guidance from static analysis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(6):952–965, 2013.

[22] M. Hicks, C. Sturton, S. T. King, and J. M. Smith. SPECS: A lightweight runtime mechanism for protecting software from security-critical processor bugs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 517–529, Istanbul, Turkey, 2015. ACM.

[23] D. Lampret. OpenRISC 1200 IP core specification, 2001.

[24] S. Ma and J. Huang. Penalized feature selection and classification in bioinformatics. *Briefings in bioinformatics*, 9(5): 392–403, 2008.

[25] A. Meixner and D. J. Sorin. Detouring: Translating software to circumvent hard faults in simple cores. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 80–89, June 2008.

[26] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 361–377, New York, NY, USA, 2015. ACM.

[27] S. Narayanasamy, B. Carneal, and B. Calder. Patching processor design errors. In *2006 International Conference on Computer Design*, pages 491–498, Oct 2006.

[28] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[29] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas. Patching processor design errors with programmable hardware. *IEEE Micro*, 27(1):12–25, Jan. 2007.

[30] S. R. Sarangi, A. Tiwari, and J. Torrellas. Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware. In *Proceedings of the 39th Annual*

IEEE/ACM International Symposium on Microarchitecture, MICRO 39, pages 26–37, Washington, DC, USA, 2006. IEEE Computer Society.

[31] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically inferring security specifications and detecting violations. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 379–394, Berkeley, CA, USA, 2008. USENIX Association.

[32] S. G. Tucker. Microprogram control for System/360. *IBM Systems Journal*, 6(4):222–241, 1967.

[33] F. Yamaguchi, F. Lindner, and K. Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[34] H. Zou and T. Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.