SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs

Matthew Hicks
University of Michigan
mdhicks@umich.edu

Cynthia Sturton
University of North Carolina at Chapel Hill
csturton@cs.unc.edu

Samuel T. King
Twitter, Inc.
sking@twitter.com

Jonathan M. Smith
University of Pennsylvania
jms@cis.upenn.edu

Abstract

Processor implementation errata remain a problem, and worse, a subset of these bugs are security-critical. We classified 7 years of errata from recent commercial processors to understand the magnitude and severity of this problem, and found that of 301 errata analyzed, 28 are security-critical.

We propose the SECURITY-CRITICAL PROCESSOR ERRATA CATCHING SYSTEM (SPECS) as a low-overhead solution to this problem. SPECS employs a dynamic verification strategy that is made lightweight by limiting protection to only security-critical processor state. As a proof-of-concept, we implement a hardware prototype of SPECS in an open source processor. Using this prototype, we evaluate SPECS against a set of 14 bugs inspired by the types of security-critical errata we discovered in the classification phase. The evaluation shows that SPECS is 86% effective as a defense when deployed using only ISA-level state; incurs less than 5% area and power overhead; and has no software run-time overhead.

Categories and Subject Descriptors C.0 [*General*]: Hardware/software interfaces; C.0 [*General*]: System architectures; K.6.5 [*Security and Protection*]: Invasive software (e.g., viruses, worms, Trojan horses)

Keywords Processor errata; hardware security exploits; security-critical processor errata

1. Introduction

Modern processors are imperfect. Processors are built from millions of lines of code [42], yielding chips with billions of transistors [22]. Verifying the functional correctness of hardware at such scales remains intractable. The resulting gaps in functional verification mean that today's production hardware typically ships with bugs. For example, the errata document for Intel's Core 2 Duo processor family [20] lists 129 *known* bugs.

	Catch All Bugs?	Catch Security-Critical Bugs?	Low Overhead?
SW-only	✗	✗	✓
HW-only	✓	✓	✗
SPECS	✗	✓	✓
SPECS+SW	✓	✓	✓

Table 1: The design space for catching processor bugs: existing software-only approaches are limited, but practical; existing hardware-only approaches are powerful, but impractical; and SPECS, combined with existing software approaches, is both powerful and practical.

Some processor bugs have the potential to weaken the security of software by allowing unprivileged code to modify or control privileged processor state. We deem these bugs *security critical* (we do not claim that this set is unique). While some effects of bugs are detectable by software through periodic checks [8, 27] or special encodings [6, 40], other processor bugs affect software in a manner that software cannot detect practically. Such bugs are security-critical because they affect a critical subset of processor functionality trusted by the software tasked with detecting and recovering from imperfections. One example of a security-critical bug is a design flaw in the MIPS R4000 processor that allows user-mode code to control the location of an exception vector [26]. The effect of this bug is user-mode code that runs at the supervisor level—privilege escalation.

These types of flaws compromise otherwise correct software—especially software that implements security policies. There are many documented examples [4, 5, 29–32] of the impact security-critical bugs have on software. Software-only patching approaches have been explored (e.g., microcode [16, 43], re-compilation [28], and binary translation [35]), but are not able to address all processor bugs [33]. Further, many of the existing patching mechanisms incur large performance penalties [38, 39], mainly due to the coarse granularity and heavyweight nature of their consistency checks.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.

ACM 978-1-4503-2835-7/15/03.
http://dx.doi.org/10.1145/2694344.2694366

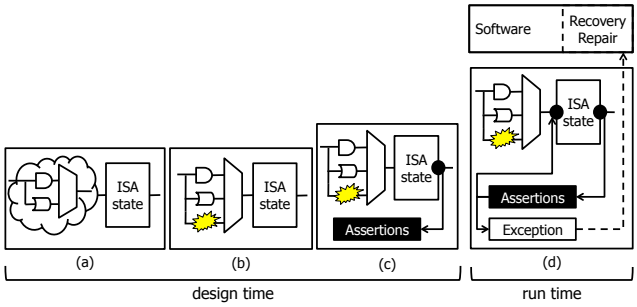

Figure 1: Processor design flow with SPECS: (a) Hardware description language implementation of the instruction set (b) At some point during design, a bug is introduced. (c) Assertions that implement SPECS invariants are added to the processor, with taps directly on the outputs of ISA state storage elements. (d) To squash in-flight state, SPECS adds write gating to the inputs of ISA state storing elements and attaches the result of dynamic verification to the exception logic, which triggers existing recovery/repair software in the event of an invariant violation.

Alternatively, hardware-based bug defenses can address all processor bugs using checks on every cycle, but this comprehensiveness comes at a prohibitive cost in terms of hardware area and power. One example illustrating the tradeoffs of hardware patching mechanisms is Diva [1]. Diva relies on modular redundancy; a second formally-verified but lower-performance processor core checks the calculations of the buggy, high-performance core. Diva can protect against all processor bugs, but is complex (potentially creating new processor bugs) and has a high hardware cost. A second example is Phoenix [37], which detects potentially inconsistent processor state by comparing the hardware-level state of the processor to known buggy state values; Phoenix requires more hardware than Diva and cannot respond to unknown bugs.

Rather than incur the costs of protecting software from all processor bugs, we propose a hybrid approach, shown in Table 1: SPECS adds a small amount of additional hardware to the processor that dynamically verifies a subset of processor functionality. The goal is to protect software from security-critical processor bugs, especially those that it cannot protect itself from efficiently. SPECS works by enforcing instruction-set-specification-derived invariants on security-critical processor state dynamically. The invariants are enforced by a series of simple assertions in hardware (Figure 1(c)). The assertions analyze ISA-level state and events to validate in-flight (i.e., not yet visible at the software level) ISA-level state updates. If in-flight state violates a SPECS invariant, a combination of assertions fire that then triggers an exception that squashes the inconsistent in-flight state (Figure 1(d)). From here, existing software recovery/repair mechanisms take over—while SPECS continues to work in the background. Essentially, SPECS acts as an interposition layer for a range of existing processor repair and recovery approaches that allows software to execute past security-critical bugs safely.

AMD ID#	Synopsis	Class	Sub class
418	Incorrect translation tables used		
561	Incorrect page walks	MA	
731	Using large pages causes incorrect IO address translations		
77	Using unaligned descriptor table allows software to jump past the end of the table without causing an exception	XR	
170	May use instructions from old page tables pointed to by old CR3		
578	Branch prediction causes invalid control flow	EI	
639	CALL instruction treated as a NOP		
776	Branch prediction causes invalid control flow		
144	Shadow RAM not invalidated	IR	
784	Control register data leaked through load operation		
165	Guest can clear data in VMCB		
503	Writes to APIC task priority register use the wrong register	IU	IL
573	Instruction pointer updated incorrectly		
734	Incorrect data stored to and past VMCB		
770	Privilege escalation		
171	Execute breakpoint handler with mixed guest and host state		
248	TLB pages not invalidated		
342	Interrupts disabled for guest		
385	Incorrect error address reported		
401	Hypervisor doesn't get correct IP		
440	CR3 not saved correctly	IU	IX
563	Flags not stored correctly in VMCB		
564	Auto halt flag not set in SMM save state		
659	VMCB flag bit not cleared		
691	Cache not invalidated correctly		
704	Incorrect IP stored		
738	Incorrect SP stored in VMCB		
744	A CC6 transition may not restore trap registers		

Table 2: Security-critical errata mined from AMD processors manufactured between 2006 and 2013. The security-critical errata fit into five classes: MA (arbitrary memory access), XR (exception related), EI (execute incorrect instruction), IR (incorrect results), and IU (invalid register update). Further, there are two types of IU: IL (illegal) and IX (incorrect).

We make three contributions in this paper:

- We identify and classify security-critical errata from recent commercial processors and discover that these bugs involve invalid updates to privileged ISA-level state, which is updated in a few simple ways (Section 2).
- We introduce a lightweight dynamic-verification strategy against security-critical processor bugs, providing an introspection point whenever security-critical state is updated outside the specification (Section 3).
- We implement (Section 4) and evaluate (Section 5) SPECS, showing that it is possible to protect software from a range of realistic security-critical processor bugs. The evaluation shows that SPECS: (1) is able to detect 86% of the implemented processor bugs using only ISA-level state; (2) requires less than 5% additional hardware; and (3) has no software run-time overhead. [1]

[1] We make the errata used for our analysis and our hardware and software source code available publicly [19].

2. Security-Critical Errata

Existing work on processor bugs as security vulnerabilities takes the approach of finding a single processor bug and realizing a usable software-level attack with that bug as a foothold [3, 11, 12, 23]. While this level of analysis highlights the threat of a specific security-critical processor bug, it does not give an idea of how pervasive security-critical bugs are, their range of effects, or what they have in common. We take a wider approach in an attempt to learn about security-critical bugs in general and to guide evaluation of our detection mechanism SPECS.

We took a systematic approach to identifying security vulnerabilities in processors. First, we collected the errata documents from the most popular commercial desktop, mobile, and embedded processors. To place a reasonable limit on this work, we limited ourselves to errata documents covering processors released in the last seven years (i.e., since 2007). For each errata document, we read the vendor's description of each erratum and determine if it represents a bug that *can* be used to allow unprivileged software to gain access to or control the behavior of privileged state of the processor in a way that contradicts the ISA. If the erratum is a possible attack foothold, we mark it as security critical. When errata documents are unclear about how software triggers the erratum, we mark it as security critical if, and only if, we believe it *possible* for a user process or guest OS to trigger it.

Table 2 gives the results from our classification of AMD errata. From a total of 301 AMD errata, we found 28 to be security critical, and split these into 5 classes: IU (invalid register update) (64%), EI (execute incorrect instruction) (14%), MA (arbitrary memory access) (11%), IR (incorrect results) (7%), XR (exception related) (4%). Figure 2 presents a visualization of the distribution of errata into the five classes.

Since it is not immediately clear from the class names what properties of a security-critical bug cause it to be in that class, here is an example bug for each class:

IU - invalid register update: There are two sub-classes of IU bugs: IL and IX. Regardless of sub-class, this class of bug is marked by the contamination of a privileged ISA-level register. The sub-class IL refers to contaminations that low-privilege software can use to bypass security policies implemented by higher-privileged software (e.g., if a user process could execute a sequence of instructions that caused the processor to execute in supervisor mode). The sub-class IX refers to when a processor bug causes security-related commands from privileged software to be essentially ignored. From the perspective of SPECS these two sub-classes are the same because SPECS focuses on the state itself, agnostic of how it became contaminated.

EI - execute incorrect insn: This class of bug is marked by the processor executing a different instruction than a perfect processor would. In this class, the processor correctly executes the instruction, but somewhere earlier in the pipeline, the instruction itself was contaminated. This is usually due to incorrect control flows caused by the processor bug.

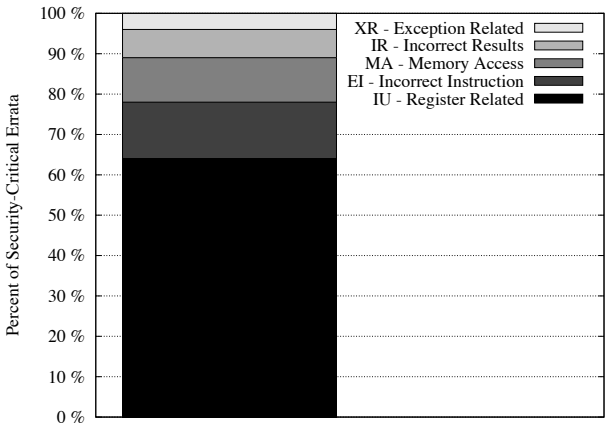

Figure 2: Class-wise composition of the security-critical errata.

MA - memory access: This class of bug is marked by the ability for any software to access memory that it should not have access to given the specification. This includes memory caches, but does not include valid accesses to shared memory buffers that were not cleared correctly (see IR).

IR - incorrect results: This class of bug is marked by an operation returning an incorrect result or not correctly updating the state (micro-architectural or ISA) of the processor correctly given the executed instruction.

XR - exception related: This class of bug is marked by an exception being ignored when it should have been handled, by the handling of an exception that should have been ignored, or by passing control to the wrong exception handler.

3. SPECS Design

SECURITY-CRITICAL PROCESSOR ERRATA CATCHING SYSTEM (SPECS) is a lightweight, hardware-based approach to protecting software from the processor bugs that can compromise system security. SPECS protects against bugs that affect reads and writes of privileged processor state. Included in this set are the bugs that software-only approaches are not be able to protect against. To detect processor bugs, it is necessary to know when and in what way processor state changes. Software-only approaches necessarily rely on privileged state as a key component of their consistency check implementation. When a security-critical processor bug compromises this privileged state, then so to are the consistency checks. Furthermore, SPECS provides a precise introspection point when privileged state changes, without which software must rely on polling or other, more coarse grain, introspection points (e.g., page faults); both decrease common case performance and leave gaps that make the system vulnerable.

SPECS catches security-critical processor bugs precisely by enforcing a select set of invariants dynamically. SPECS invariants are properties over architecturally visible processor state and events (i.e., ISA-state). They are derived from the instruction set specification and govern how and when updates to privileged processor state can occur. An example of the type of bug that SPECS will catch is privilege escalation: a change in processor mode from low to high privilege that is not consistent with the instruction set specification. Privilege escalation bugs have occurred in recent commercial processors (see Section 2).

3.1 An example invariant and assertion

We use the privilege escalation bug to help illustrate the concepts of a SPECS invariant and the implementation of that invariant using a combination of simple assertions. To start, we describe privilege escalation as a violation of the following invariant: $I_0 \doteq$ **A change in processor mode from low privilege to high privilege is caused only by an exception or a reset.**

Invariant I_0 is a statement that the instruction set specification says must be true of the system at all points of execution.

In SPECS, invariants are enforced by composing (with Boolean operations) one or more simple assertions on ISA-level processor state. For example, SPECS enforces I_0 in the following way:

$$\begin{aligned} A_0 \doteq assert(&risingEdge(\texttt{SR[SM]}) \rightarrow (\texttt{NPC}[31:12] = 0) \wedge \\ &risingEdge(\texttt{SR[SM]}) \rightarrow (\texttt{NPC}[7:0] = 0) \vee \\ &risingEdge(\texttt{SR[SM]}) \rightarrow (\texttt{reset} = 1)). \end{aligned}$$

`SR[SM]` represents the supervisor mode bit of the processor's status register and an exception is indicated by the next program counter, `NPC`, pointing to an exception vector start address. In our implementation (see Section 5), the exception vector start address is of the form `0x00000X00` (the "X" indicates a don't-care value).[2]

3.2 Design principles

The high-level design of SPECS follows three principles;

1. *Maintain current ISA abstractions.* SPECS should detect processor imperfections automatically, without changes to existing software.
2. *Keep the common case fast.* SPECS should only interrupt software in the rare case that a processor violates one of SPECS's security invariants—precise introspection.

[2] This might seem as if it leaves the door open for a processor attack that escalates privilege while executing at an address that matches the form `0x00000X00`, but it does not. Pages in that address range have supervisor permissions set which implies that code executing in that address range is already in supervisor mode. If software that exercises a security-critical processor bug attempts to allow user mode execution of supervisor mode pages, SPECS includes an invariant to detect such misbehavior.

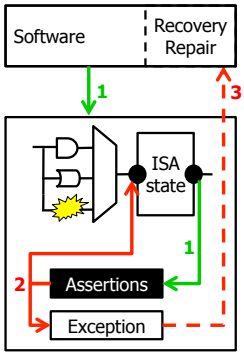

Figure 3: Flow of detection and recovery in SPECS: 1) Software executes normally on the processor while invariants implemented using hardware assertions check the processor's state for specification violations. 2) A violation of a SPECS invariant triggers an exception and prevents state updates. 3) The exception causes any in-flight (i.e., contaminated) state to be flushed and control passes to existing recovery/repair software. 1) SPECS continues to protect the recovery/repair software.

3. *Be practical.* To maximize practicality, we aim to implement invariants using only ISA-level state. The ISA is more stable across processor families and less complex than micro-architectural state. Additionally, SPECS implements invariants using improved versions of industry-standard assertions.

3.3 Assumptions

SPECS relies on three assumptions. The first is that the instruction set specification is correct. Ambiguities in the specification lead to implementation differences, which lead to security vulnerabilities [45]. For SPECS, the specification drives the design and implementation of the invariants; thus, specification ambiguities can lead to false detections or worse, missed detections.

The second assumption is that it is possible to dynamically verify the correctness of an ISA-level state update using only the current ISA-level state and the instruction. This forms the basis of our approach: SPECS invariants are defined over the proposed ISA-level state update, the current instruction, and the current ISA-level state. When this assumption does not hold, we have to implement security invariants using low-level state (see Section 5 for one exception) or risk missing bug activations.

Finally, the focus of our work is the pipeline core of the processor; we assume the memory hierarchy is correct. Note that we do not foresee any fundamental limitations to applying the SPECS approach to other units in a processor. Our experience indicates that the SPECS approach works well where it is possible to precisely define expected behavior and where there is a subset of critical behaviors that need protecting.

3.4 SPECS components and interactions

Figure 3 shows the key components of SPECS and their interactions. Software executes normally on the processor

while assertions in hardware check processor ISA-level state and events for specification violations. The assertions verify that a proposed ISA-level state update is valid—i.e., does not violate an invariant—for security-critical state given the instruction and the current (already verified) ISA-level state. If the proposed ISA-level state update is valid, then: (1) no combination of assertions fire to signify an invariant violation; (2) the processor is allowed to commit the proposed state to the ISA level; and (3) execution continues. On the other hand, if a state update violates an invariant, SPECS triggers an exception causing any contaminated in-flight state to be flushed. From this point, existing recovery/repair software can take control as required to push the state of software forward. Note that SPECS continues to protect the recovery/repair software; this added safety requires that any recovery/repair approach be able to handle recursive detections.

3.5 History

Some invariants require more history than the current and proposed ISA-level state provide. For invariants over control flow (Section 4) we need to check previous instructions for valid causes of control flow discontinuities. Valid causes of control flow discontinuities in our implementation are branches and jumps; return from exception; system calls and traps; and exceptions. For exceptions and system calls/traps (which we treat as exceptions) the assertion logic doesn't need any extra history information because the current PC informs us whether we got here due to one of the listed events. For this, we look to see if we are at an address associated with a exception handler. For return from exception instructions, we need to know the instruction before the break in control flow. The remaining control flow disrupting events require the ability to see the instruction associated with the current (most recently verified) ISA-level state to verify that it was a jump or branch. Making the history problem worse is an architectural feature known as a branch delay slot. In many architectures, including the one we implement SPECS in, the processor always executes the instruction directly after the branch/jump instruction; thus the instruction after jumps/branches is also executed. The goal of the branch delay slot is to reduce the number of pipeline bubbles due to breaks in control flow. This means that when SPECS's continuous control flow invariant gets violated, we have to look back two instructions previous for a branch or jump instruction.

There are two ways to solve this problem: we can store the previous two instructions or we can store—and use in our assertions—the previous two assertion results. We choose to implement the latter since it requires two extra flip-flops for each micro assertion that is impacted by a delay slot and one for those that are not (7 total), since the result of all assertions is a single bit. In comparison, keeping the previous two instructions around would require 64 additional flip-flops.

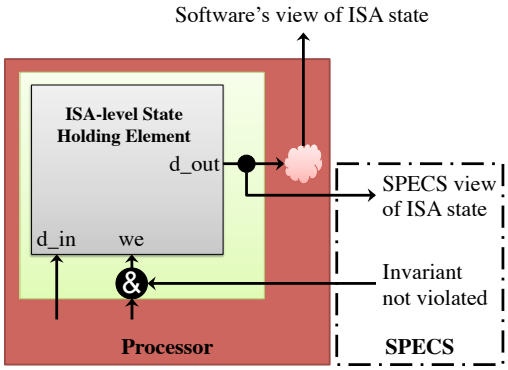

Figure 4: Low-level hardware support required for SPECS: SPECS requires direct access to state holding elements (i.e., before any other circuit manipulates the value) and requires the ability to prevent updates to ISA-level state in the event of invariant violations. Without either of these, SPECS misses bugs in our evaluation platform.

3.6 Instruction stability

A preliminary evaluation showed that SPECS missed security-critical bugs that changed the instruction in the pipeline. For example, a security-critical processor bug could change a privilege de-escalation into a no-op, causing it to be ignored and resulting in a privileged escalation. SPECS misses such bugs because it must trust the instruction as a state input. Additionally, there is only a single instruction at the ISA-level (SPECS reads the decode stage version), but in reality each pipeline stage may see a different instruction.

To ensure that we can trust that SPECS's view of the instruction is what the pipeline sees, we add a micro-architectural-level invariant that enforces instruction integrity as it moves through the pipeline. The invariant states that the instruction loaded from memory is seen at each pipeline stage. Refer to Section 4 for detailed information on this invariant.

3.7 Local state taps

It is critical that the assertions get their view of ISA-level state from as close to the state holding element as possible. Any logic between the state holding element and SPECS's state taps is vulnerable to bugs. In fact, in the first SPECS prototype, we missed catching bugs (bug 5) due to buggy logic between the state taps and the state holding element. Figure 4 shows our general approach to solving this problem. We find the actual memory that holds the state value, and read the state directly from that memory's output. Often times, this reduces to inserting SPECS's state taps between the state holding element and the rest of the processor.

3.8 Invariant state update gating

Similar to where we place SPECS's state taps, we found that we need to add write gates directly to the inputs of state holding elements. It is essential that SPECS squashes in-flight state when invariants are violated. We do this is by guarding writes at each privileged ISA-level state element using the

ID	SPECS Invariant	OVL Implementation
1	Execution privilege matches page privilege	always((SR_SM & MMU_SXE) || ($\overline{SR\_SM}$ & MMU_UXE))
2	SPR = GPR in register move instructions	posedge(INSN = l.MTSPR, SPR = GPR)
3	Updates to exception registers make sense	posedge((PC & 0xFFFFF0FF) = 0, (EEAR = EA) && (EPCR = SR[DSX] ? PPC-4 : PPC) && (ESR = (SR & 0xFFFFDFFF)))
4	Destination matches the target	posedge(GPR_WRITTEN, GPR_TARGET = (INSN & TARGET_MASK))
5	Memory value in = register value out	posedge((INSN = l.LWZ) || (INSN = l.LHZ) || (INSN = l.LHS) || (INSN = l.LBZ) || (INSN = l.LBS), GPR = MEM_BUS)
6	Register value in = memory value out	posedge((INSN = l.SW) || (INSN = l.SH) || (INSN = l.SB), GPR = MEM_BUS)
7	Memory address = effective address	posedge((INSN = l.SW) || (INSN = l.SH) || (INSN = l.SB) || (INSN = l.LWZ) || (INSN = l.LHZ) || (INSN = l.LHS) || (INSN = l.LBZ) || (INSN = l.LBS), ADDR_CPU = ADDR_BUS)
8	Privilege escalates correctly	posedge(SR[‘OR1200_SR_SM], (RST = 1) || (PC = 0x00000X00))
9	Privilege deescalates correctly	posedge(INSN = l.RFE, SR[‘OR1200_SR_SM] = ESR[‘OR1200_SR_SM]) && posedge((INSN = l.MTSPR) && (INSN_target = SR), SR[‘OR1200_SR_SM] = GPR_SOURCE[‘OR1200_SR_SM])
10	Jumps update the PC correctly	next((INSN = JMP) || (INSN = BR), PC = EA, 2)
11	Jumps update the LR correctly	next((INSN = JMPL) || (INSN = JMPLR), LR = PPC+4, 2)
12	Instruction is in a valid format	always((INSN & Class_Mask) = Class) && ((INSN & Reserved_Mask) = 0)
13	Continuous Control Flow	delta(PC, 4, 4) || assert((INSN = JMP) || (INSN = BR) || (INSN = RFE)) || assert((PC & 0xFFFFF0FF) = 0)
14	Exception return updates state correctly	next(INSN = l.RFE, (SR = ESR) && (PC = EPCR), 1)
15	Reg change implies that it is the instruction target	(posedge(GPR_Written, (INSN & OPCODE_MASK) = (20-2F, 06, 38)) && posedge(GPR_Written, (INSN & TARGET_MASK) = GPR_Written_Addr)) || posedge(GPR9_Written, ((INSN & OPCODE_MASK) = JAL) || ((INSN & OPCODE_MASK) = JALR))
16	SR is not written to a GPR in user mode	posedge(GPR_WRITTEN, GPR_TARGET $\neq$ SR)
17	Interrupt implies handled	next((INSN & 0xFFFF0000) = 0x20000000, ((PC & 0x00000F00) = 0xE00) || ((PC & 0x00000F00) = 0xC00), 1)
18	Instruction not changed in the pipeline	next($\overline{IF\_flush}$ & ICPU_ack & $\overline{IF\_freeze}$, INSN_F = INSN_MEM) || next($\overline{ID\_freeze}$, INSN = INSN_F) || next($\overline{EX\_freeze}$, INSN_E = INSN)

Table 3: Invariants developed to protect against security-critical processor bugs: The first 14 invariants were developed by only looking at the specification. The next three invariants were added in response to what we learned in classifying existing commercial processor errata (Section 2). The final invariant was added after one of our errata-based bugs evaded the initial SPECS implementation (Section 5).

result of SPECS's invariants. Figure 4 shows how we add write gating: we add an AND gate to the pre-existing write enable signal that controls updates to the state item. The AND gate takes the original write enable and a signal which takes the value zero whenever any invariant fires. Write gating causes writes to any protected state to be ignored if the current execution violates any SPECS invariant.

4. SPECS Invariants

This section details the SPECS invariants for our OR1200-based prototype.[3] Table 3 contains an identification number, concise description, and assertion-level implementation for each SPECS invariant. These invariants are *not* ad hoc: 1–14 come directly from the specification, 15–17 come from our analysis of AMD errata, and 18 comes from our preliminary evaluation. Additionally, there are fewer errata than there are security-critical errata because many security-critical have the same effect (i.e., they are in the same class) even though the activation mechanism is different and SPECS is concerned only with effect.

[3] Since we do not have access to the source code for any AMD processor covered by our errata analysis, we experiment on the OR1200. The OR1200 is a RISC processor with a Harvard architecture, five-stage pipeline, memory management unit, and caches—comparable to a mid-range mobile part.

4.1 Assertions

Before tackling the invariant descriptions, it is important to understand the behavior of the assertions we construct them out of. In keeping with our design goals, we modify industry standard Open Verification Library (OVL) assertions [14] so that they are synthesizable (originally, some were simulation-only). We are able to construct all 18 of the SPECS invariants using only 4 of the more simple OVL assertions:

- *always*(`expression`): `expression` must always be true
- *edge*(`trigger`, `expression`): `expression` must be true when the `trigger` goes from 0 to 1
- *next*(`trigger`, `expression`, `cycles`): `expression` must be true `cycles` instruction clock ticks after `trigger` goes from 0 to 1
- *delta*(`signal`, `min`, `max`): when `signal` changes value, the difference must be between `min` and `max`, inclusive

4.2 Invariant descriptions

Table 4 contains written descriptions of each invariant in Table 3 and provides details of the OR1200 as needed. It is numbered to coincide with the ID listed in Table 3.

ID	SPECS Invariant Description
1	The privilege of the memory page the current instruction comes from matches the privilege of the processor.
2	Instructions that load a special-purpose register (privileged) with a value from a general-purpose register load do not modify the general-purpose register value.
3	The OR1200 has three registers that save the state of software when an exception occurs. The EPCR stores the PC at the time of the exception, the ESR stores the status register (SR), and the EEAR stores the effective address at the time of the exception. This invariant fires when any of these exception registers are not updated correctly.
4	The register update as the result of executing an instruction is the register specified as the target register by the instruction.
5	In memory loads, the value stored in the target register is exactly the value from the memory subsystem.
6	In memory stores, the value sent to the memory subsystem is exactly the value of the register specified in the store instruction.
7	The address sent to the memory subsystem is exactly the effective address given the GPR values and instruction contents (i.e., addressing mode and immediate).
8	If the execution privilege, stored in SR, goes from 0 (user mode) to 1 (supervisor mode), then it must be the result of taking an exception or a processor reset.
9	The execution privilege goes from 1 to 0 when a value with a 0 in the mode bit position is loaded into SR or when a return from exception is executed and the mode bit in the ESR is 0.
10	Branch and jump instructions generate the correct effective address and that effective address is loaded correctly into the program counter (PC).
11	Jump and link instructions store the address of the instruction immediately following the delay slot instruction to the link register (LR).
12	The reserved bits of a given instruction are set to 0 for each instruction encoding class
13	The address of the current instruction is the address of the previous instruction plus four. This invariant is a building block used to trigger other invariants that verify control flow discontinuities.
14	The return from exception instruction causes the PC to be loaded from EPCR and the SR to be loaded from ESR.
15	When a register changes, it must be specified as the target of the instruction.
16	When a target unprivileged register changes, the value written to that register is not equal to the SR.
17	When a software created interrupt occurs the processor passes control to the appropriate exception handler. In the OR1200, the exception handlers are at fixed address in the start of the system's address space.
18	Once the instruction fetch stage latches a new instruction, that instruction stays the same as it transitions between pipeline changes. In the OR1200, the instruction is not needed by the memory or write-back stages, so the invariant only checks up to the execute stage. Additionally, the OR1200 squashes mid-pipeline instructions by changing the instruction to a special no-op instruction, so we have to gate invariant 18 on a check to see if the instruction changed to this special no-op.

Table 4: Detailed descriptions of invariants; IDs correspond to the invariant IDs in Table 3.

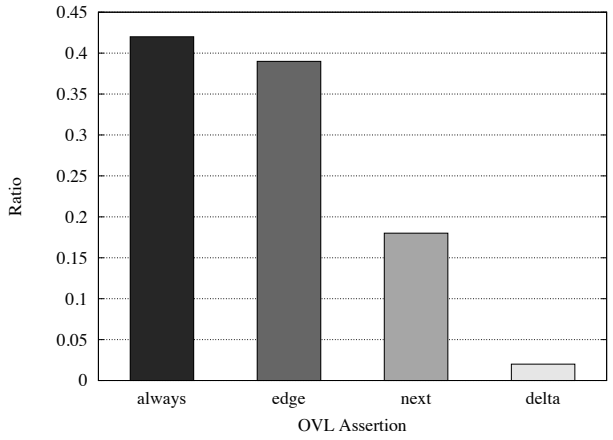

Figure 5: The proportion of each OVL assertion that we use in implementing SPECS invariants. The OVL assertions are ordered left to right in terms of increasing hardware complexity.

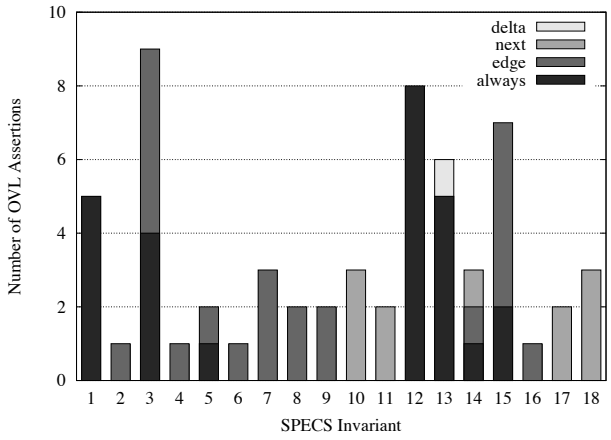

Figure 6: For each SPECS invariant, the number and type of OVL assertions required to implement the invariant and the composition of OVL assertions.

4.3 Invariant analysis

Figure 5 shows that the most common OVL assertion is *always*, followed closely by *edge*. *delta* is used only once. One observation is that the use of an assertion is inversely proportional to its complexity: we use the simplest assertions most often. Another observation is that we can implement all of our invariants using only 4 of the 33 assertions provided by OVL.

Figure 6 shows how many of each OVL assertion it takes to construct each SPECS invariant. 13 of the 18 invariants use only a single type of assertion. For the invariants that require multiple types of assertions, there is no clear pattern

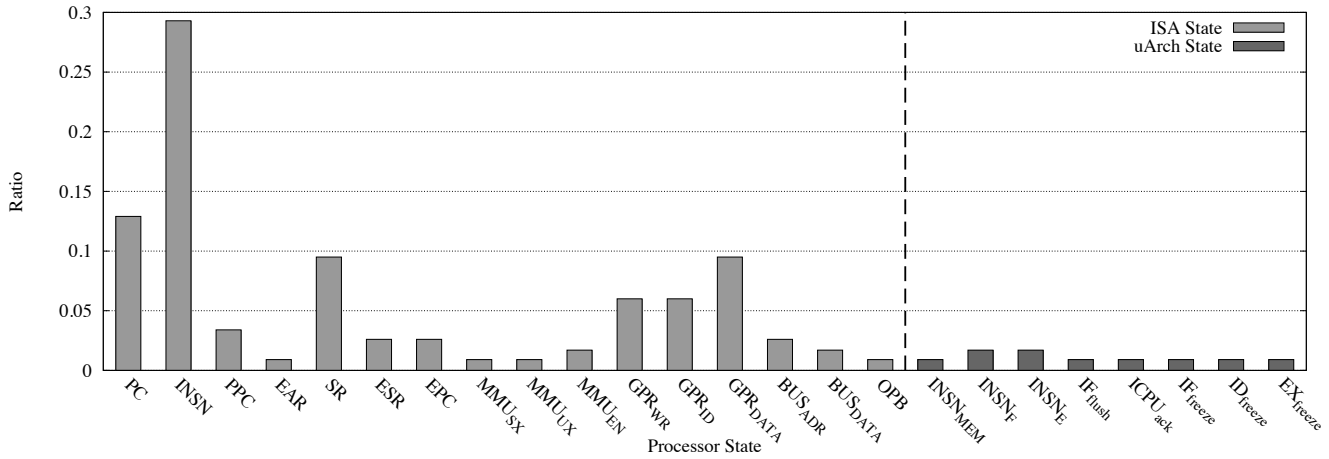

Figure 7: The ISA and micro-architectural state required to implement the 18 SPECS invariants and a tally of how many times each state item is referenced.

on which types of assertions tend to go together. The most complex invariants use more assertions, but are no more diverse with respect to assertion type than simple invariants.

Figure 7 shows the ISA and micro-architectural state that SPECS invariants rely on when dynamically verifying ISA-level state updates. The most important state item is the instruction (INSN) itself—which is why we added invariant 18 to protect the instruction as it moves through the pipeline. The other important state items are the program counter (PC) and information about register file updates, i.e., is this a write (GPR_WR), what register are we updating (GPR_ID), and what value is being written (GPR_DATA). As it resides outside of the register file, the status register (SR)—which contains the mode bit—is also frequently used by SPECS invariants. To get an idea of what SPECS looks like in an AMD-class processor, look at the difference between uses of SR and ESR (the exception-backup of SR). SR is involved in three times more invariants than ESR even though the both are privileged and hold roughly the same data. The difference in popularity comes from the ability to directly manipulate the SR using software-level instructions, while the only way to change the value of ESR is to trigger an exception. We expect this trend to hold for processors of any complexity: the more ways to update security-critical state, the more invariants required to protect that state.

Another point to note is that the one micro-architectural-level invariant tends to use more state than the ISA-level invariants. A wider footprint makes the invariant more complex and less portable.

5. Evaluation

To demonstrate that SPECS can detect and recover from a realistic and diverse set of security-critical processor bugs, we built a prototype implementation in the OR1200 processor [34] and tested it against 14 processor bugs modeled after the classes of errata from Figure 2 and associated bug-activating programs.[4] We choose the OR1200 processor because it is an open source processor implementation of non-trivial complexity that allows us to build and evaluate our SPECS design. Ideally, we would implement SPECS in an AMD processor, but complete, open source implementations are not available for x86/x64 architectures. The OpenRISC OR1200 processor is a mature, open source, 32-bit, RISC processor implementing the OR1K instruction set. It has a five stage pipeline, separate instruction and data caches, and support for virtual memory. These traits make it popular in research prototypes and it has even been used in commercial products [2, 21, 36]. Inside the OR1200, we craft each of the 14 buggy systems to model the security-critical bug classes that we mined from recent commercial processor errata (Section 2); this ensures that our test cases represent realistic design defects. For each buggy system, we create a program that activates the bug, violating some security policy.

For our prototype, we implemented the 18 invariants described in Table 3. The majority of the invariants, 14, come directly from our analysis of the specification, 3 come from the results of our AMD errata analysis, and the last invariant comes from a preliminary evaluation of our SPECS prototype. Of the 18 invariants, 17 keep with our goal of using only ISA-level state.

The goal of this evaluation is to show that SPECS (1) is versatile enough to detect the activations of a wide variety of security-critical processor bugs; (2) has low hardware and software overheads; and (3) supports existing software-only recovery and repair mechanisms.

5.1 Detecting the errata-based bugs

As shown in Table 5, SPECS assertions correctly detect all of the errata-based buggy systems. SPECS detects 12 of the 14 bugs (87%) using only ISA-level state. The two

[4] There is no relationship between the AMD, Bug, and Invariant ID numbers.

Bug ID	Synopsis	Class	Detected	Pre-recovery Cleanup
1	Privilege escalation by direct access	IU	✓	Correction
2	Privilege escalation by exception	IU	✓	Correction
3	Privilege anti-de-escalation	IU	✓	Correction
4	Register target redirection	IU	✓	Correction
5	Register source redirection	IU	✓	None
6	ROP by early kernel exit	IU	✓+	Backtrack
7	Disable interrupts by SR contamination	IU	✓	None
8	EEAR contamination	IU	✓	Correction
9	EPCR contamination on exception entry (from PC)	IU	✓	Correction
10	EPCR contamination on exception exit (to PC)	IU	✓	None
11	Code injection into kernel	EI	✓	Backtrack
12	Selective function skip	EI	✓+	Backtrack
13	Register source redirection	IR	✓	None
14	Disable interrupts via micro arch	XR	✓	Backtrack

Table 5: Errata-based buggy systems: For each errata-based bug, a description of its effect, its class (from Section 2), whether SPECS can detect the bug (Detected: ✓= yes, + = requires micro-architectural state), and the expected minimum recovery required after SPECS squashes in-flight state: in order of complexity, None: no special action required; Correction: need to reset some ISA state to a default/safe value; and Backtrack: need to rollback some ISA state to its previous value.

bugs that bypass the ISA-only invariants do so in the same way: they change the instruction as it moves through the pipeline. This bypasses SPECS because its assertions trust that the instruction is correct. To detect this type of bug we define an invariant using micro-architectural-level state that ensures that the instruction the ISA-level assertions see is the instruction that was loaded from memory. Adding a single invariant (#18) to ensure instruction integrity is enough to detect the remaining two bugs (bugs 6 and 12).

5.2 Protection versus time

We believe that because SPECS focuses on the effect of the problem as opposed to the cause of the problem, processor designers will incorporate SPECS in a model where security-critical bugs from previous generation processors become SPECS invariants in next generation processors. To support this point, we investigate how SPECS protects later processors by looking at the invariants required to protect earlier processors. For this, we add invariants in the order of the AMD bug that motivated them. The first 14 invariants came directly from the specification, before analyzing any errata. With just these 14 invariants, SPECS detects 7 of the 14 bugs we implement and the chronologically first 11 of the 28 security-critical errata from recent AMD processors (39%). After adding an invariant to detect activations of the 12^{th} AMD bug, SPECS detects 12 of the 14 buggy systems and addresses the first 15 of the 28 AMD bugs (54%). Finally, after adding an invariant to handle the 16^{th} AMD bug, SPECS detects all of our 14 bugs and the remaining AMD bugs (100%). We feel that this result supports our classification of AMD bugs into equivalence classes and provides a good model for the life-cycle of SPECS.

5.3 False detections

In our experiments, there are no false detections given the final set of SPECS invariants, but it did not start that way. When it is not possible to precisely define an invariant for a given security-critical state update, there is a tradeoff between missing bug activations and firing when no bug has been activated. SPECS supports erring on either side of the tradeoff space, but for this paper, we prefer the safety of false detections. This section presents each initial false positive and the approach used to eliminate it.

Invariant 3 can produce false detections because of the limits of implementing the invariant at the ISA level. Invariant 3 verifies that the processor updates the exception registers correctly in the event of an exception. False detections occur when an exception interrupts the first instruction in a basic block (a continuous and contiguous dynamic trace of instructions). When this occurs, there is no record of the address that should have been saved to Exception Program Counter Register (EPCR; it holds the PC to return to after the exception returns): the invariants only see the current state and the state about to be committed. Since the instruction was interrupted by the exception, its state was flushed, along with it the PC value that should be saved to EPCR. This means that invariant 3 fires even when the processor updates EPCR correctly—a false detection. We can avoid this by looking for impending exceptions in the invariant.

To determine the rate of this false detection, we run a gamut of software on our FPGA-based prototype, including Linux and MiBench [15] benchmarks. We find that the rate of false detections is approximately 1 detection per 20,000 instructions. These detections all occur in a single burst while booting Linux—which is exception heavy (the upper limit is the rate of exceptions). A deeper analysis reveals that the most common cause of false detections is jumping

	Baseline	Minimum	All
Logic	7911 LUTs	2.16%	4.26%
State	10549 Bits	2.74%	4.95%
Power	4.17 W	.17%	.58%
Delay	13.3 ns	0%	0%

Table 6: SPECS overheads: baseline for the OR1200, Xilinx xupv5-lx110t-based System-on-Chip compared to the overheads of SPECS with only the invariants required to detect all bugs in Table 5 (Minimum) and all 18 invariants listed in Table 3 (All).

to an address not in the translation lookaside buffer (TLB). The OR1200 has software-managed TLBs that handle the ensuing TLB miss which causes the effective address to be lost.

Another source of false detections is invariant 1. Invariant 1 fires in a series of bursts early during Linux boot-up, but then ceases. We find that this is due to Linux not correctly assigning page protections—an incomplete implementation of a security policy. Once Linux's boot-up routines correctly set page permissions, invariant 1 no longer fires.

5.4 Evaluating the cost of SPECS

Experiments show that SPECS effectively addresses all errata-based bugs that make up our evaluation platform. Experiments also show that false detections (and by extension, activations of recovery/repair software) are rare. The goal of this section is to measure the hardware overhead of SPECS. We look at all invariants as a group, the relative overheads of individual invariants, and the relative overheads of each OVL assertion that SPECS employs.

Table 6 shows less than 5% hardware overhead and no run-time overhead (in the common case) due to SPECS. Table 6 explores the hardware overheads at two design points: *Minimum* represents SPECS with only the invariants required to detect all of the errata-based bugs, while *All* represents all invariants being implemented in the OR1200. The invariants from Table 3 that make up *Minimum* are 3, 8, 9, 14, 15, 17, and 18.

There is a sizable jump in overhead when going from the 7 invariants in *Minimum* to all 18 invariants. Figure 8 shows how each invariant contributes to the total overhead of SPECS. Invariants 3, 5, and 13 stand out as the most costly invariants. One might assume that these three invariants are the most complex or require the most OVL assertions, but by comparing Figures 6 and 8, we see that is not always the case. For example, invariants 1 and 12 both use relatively many assertions, but they have relatively low overheads. This is because these invariants read ISA-level state that is much smaller than 32-bits. This allows the tools to optimize the assertions, reducing their gate count.

It is also difficult to determine the hardware cost of a given invariant due to the different hardware cost of the individual assertions. Figure 9 presents the hardware overheads of the OVL assertions that SPECS invariants use. Figure 9

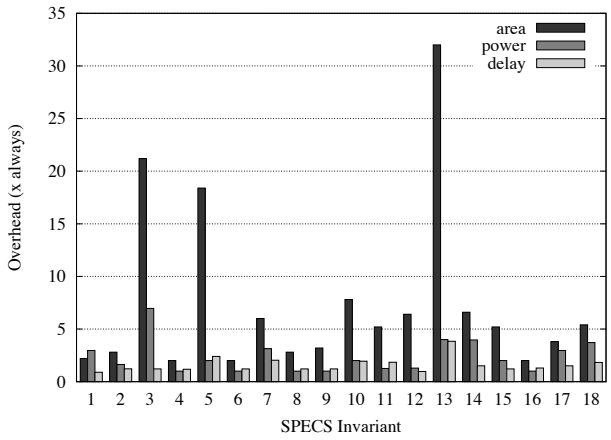

Figure 8: Hardware overheads of each SPECS invariant relative to the OVL always assertion.

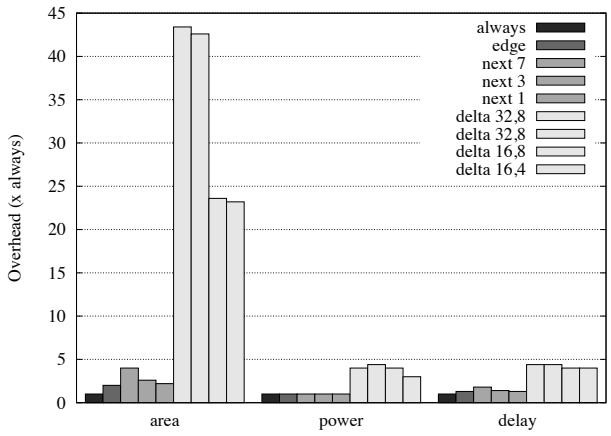

Figure 9: Hardware overheads of each OVL assertion relative to the OVL always assertion. For assertions that have configuration parameters, i.e., next and delta, we explore the effects of selecting different parameters. next x: x is the maximum number of ticks between the triggering event and the expected event. delta x,y: x is the number of bits in the input data and y is the number of bits in the maximum allowed delta (e.g., 8 bits = +/-256).

also includes a parameter exploration for the assertions that are configurable. These results show the the hardware overheads vary greatly across assertion types. Fortunately, *delta*, the most expensive assertion, is also the least used, while the most used is the simplest!

5.5 Recovering from invariant violations

SPECS supports existing software-level recovery and repair approaches by providing precise introspection points when software activates security-critical processor bugs. To aid fitting existing recovery and repair approaches to SPECS, this section provides details on the consistency of ISA-level state after SPECS squashes in-flight state updates.

Table 5 lists, for each buggy system, the amount of ISA-level state clean-up necessary post detection. Sometimes SPECS write gating is enough to ensure a consistent state,

but other times a rollback to the previous state value is required. There are three distinct levels of state clean-up. Aside from no correction being required (*None*), the simplest clean-up is *Correction*, which involves resetting the contaminated ISA-level state to a safe value (e.g., setting the processor mode to user mode in the event of a privilege escalation) and then continuing execution. The most complex clean-up is *Backtrack*. *Backtrack* differs from *Correction* in that there is no safe or obvious previous value. This means that the previous value needs to be restored by the recovery or repair software. One approach to *Backtrack* is to provide a few cycles of checkpointing for security-critical state. Another approach is to recalculate the value by retracing execution.

6. Discussion

Our presentation of SPECS's implementation and evaluation did not address two important points, namely, scaling SPECS to x86 class processors and limitations of SPECS. We address these questions now.

6.1 Implementing SPECS in x86

Section 5 shows that SPECS is effective and efficient at detecting x86-like bugs implemented in a more simple processor architecture. The longer-term goal, however, is to have SPECS adopted by commercial processor manufacturers, thus it is important to address how SPECS scales with processor complexity. Lacking an open source x86 implementation we can augment with SPECS, we address how SPECS scales in a qualitative manner, but we do expect SPECS to scale to x86 processors.

We expect SPECS to scale to x86 processors. Complexity in commercial processors stems largely from logic devoted to increasing performance—as opposed to the functionality that SPECS must protect. These optimizations add state below the level of the software interface, i.e., without changing the architecturally visible state (e.g., out-of-order execution). SPECS monitors only the architecturally visible state (e.g., special-purpose registers). This means that x86-class processors are likely to have proportionally less ISA-level state than the OR1200 we evaluate with, resulting in relatively lower overheads due to SPECS. We also expect SPECS to scale because the types of architecturally visible state and rules guarding state updates are similar in the OR1200 and x86, with the x86 just having much more state and instructions: we do not know of a unique aspect of x86 at the ISA-level fundamentally different from the OR1200.

6.2 Limitations

SPECS is not able to detect all possible processor bugs, nor can it guarantee that all inconsistent state is flushed (Section 5.5). By construction, SPECS cannot detect contaminations to general-purpose state. General-purpose state is updated in a myriad of ways; as opposed to the few and simple ways that security-critical state is updated. To apply SPECS to general-purpose functionality, the number and complexity of invariants must greatly increase. Since the assertions are at the same level of abstraction as the functional description, the increase in number and complexity of invariants is the same order of magnitude as the increase in processor functionality that SPECS must protect. While SPECS is capable, the result amounts to a second implementation of the ISA. As stated earlier, software is better suited for general-purpose protection [6].

Another limitation of SPECS is that it cannot defend against processor bugs that change ISA-level state before SPECS taps the signal. Section 3.6 showed how to use micro-architectural state to prevent the the instruction from being contaminated in this fashion (from bugs 6 and 12). Another possibility is a manufacturing defect, a single-event upset, or a wear-out that contaminates the state holding element itself or the wires that feed SPECS's view of ISA-level state. Two options to address this problem outside of SPECS are to add error correction logic to state holding memories or to manufacture security critical components of the processor at a more resilient process node.

7. Additional related work

This section covers work related to SPECS. Specifically, we cover processor errata as software-level security vulnerabilities, alternative approaches to patching design-time processor bugs, and a overview of recovery approaches and how SPECS supports them.

7.1 Processor imperfections

The idea that processor errata can impact the security of a system is not new. Theo de Raadt was the first to link Intel errata to potential security vulnerabilities in the OpenBSD operating system [11]. Inspired by de Raadt's ideas, others implemented software-level attacks that used Intel errata as a foothold [3, 12, 23]. The difference between previous efforts in exploiting a single erratum as a foothold for a software-level attack and our work is that our analysis extends beyond a single attack; we identify both a set of errata that make it past functional verification and the different classes of vulnerabilities created by these errata that software can not protect itself from.

7.2 Patching processor bugs

Ideally, all design-time processor imperfections should be addressed. The best way to do this is by proving correctness using formal verification. Although formal methods have made great strides in recent years, full formal verification of modern processors remains out of reach—as is evident from the number of errata in commercial processors [20, 24]. One approach to patching processor bugs that escape verification is to add redundant but diverse computations. DIVA [1], as mentioned in Section 1, is a processor bug patching mechanism that leverages redundant implementations of the instruction set combined inside a single processor. In DIVA, a simplified checker core verifies the computation results of

the full-featured core before the processor commits the results to the ISA level. Since the checker core must be simple enough to formally verify, its performance is reduced and slows execution; in contrast, SPECS provides introspection points that enable recovery approaches that add overhead *only* when the processor violates security invariants.

Another approach to patching arbitrary processor bugs is signature-based detection of processor bug activations. Constantinides et al. [7] and Phoenix [37] use low-level hardware state (flip-flops) to form bug signatures and monitor the run time state of the processor for matches. Such techniques can detect more imperfections than SPECS, but at a cost: (1) they require two extra flip-flops for every flip-flop in the design and (2) heavy contention on opcode flip-flops limits how many bugs the approaches can monitor concurrently [17]. These overheads can be significant; Phoenix, for example incurs up to 200% additional hardware.

Similar to signature-based detectors, but at the opposite end of the design tradeoff space is the idea of Semantic Guardians [44]. Semantic Guardians are assertions on low-level state that fire whenever the processor enters a state at runtime that was not functionally verified. This does not suffer from the heavy contention on a shared resource seen in signature-based approaches, as each unverified state has its own assertion circuit. The problem with this approach is that many states are not seen during verification and building an assertion for each state is not practical.

7.3 Recovery

The most common recovery strategy is to use a checkpointing and rollback mechanism. While there are a range of low overhead checkpointing options available [10, 13, 41], our analysis of recent errata shows that re-executing is not enough to move software state past the malicious circuit. Even in the case of a homogeneous multiprocessor, re-executing the bug-inducing instruction sequence may activate the bug. This means that a form of processor repair or recoding of software is required to handle arbitrary security invariant violations.

Changing software to recover from incomplete hardware is not a new idea; before the ubiquity of hardware floating point (FP) units, processors lacking FP hardware supported ISA-mandated FP instructions by triggering an exception and using FP emulation firmware to produce a correct result [25]. Narayanasamy et al. [33] extend this idea to processor bug patching by using simple, ad hoc, instruction rewriting routines that run on the macro expander DISE [9] in an attempt to avoid activating the bug, and suggest specific software-based recovery strategies for five errata. BlueChip [18], a further elaboration of this scheme, is a recovery mechanism targeted at malicious circuits implemented as a Linux device driver. SPECS provides a low overhead way of invoking the BlueChip recovery routines when they are needed.

8. Conclusion

SPECS is a new approach to patching processor bugs. SPECS adds a small amount of hardware to a processor that dynamically verifies security invariants. SPECS works in the background, not affecting software in the common case of a bug-free execution. Should software activate a processor bug that causes a violation of a security invariant, SPECS creates an introspection point for existing repair and recovery schemes. Experiments with a SPECS implementation show its practicality and effectiveness at detecting a range of errata-inspired processor bugs.

Our results validate the hybrid approach of SPECS. Previous research shows that software can practically address a wide range of processor bugs—but not all. Previous research also shows hardware can address the processor bugs that software cannot, but a system powerful enough to address all possible processor bugs is impractical. SPECS represents a middle ground: where small amounts of hardware ensure a baseline of processor functionality and software is free to protect itself, as it sees fit, from the bugs in the rest.

Acknowledgments

The paper was improved with constructive comments from anonymous reviewers; we appreciate your work, expertise, and insights. The work also benefited from discussions with Kevin Fu, Milo M. K. Martin, and David Wagner. This research was supported in part by Intel through the ISTC for Secure Computing, by the AFOSR under MURI Award FA9550-09-1-0539, by MARCO and DARPA via C-FAR (one of the six SRC STARnet Centers), and by the National Science Foundation under grants CCF-0810947, CNS-1331652 and CNS-1040672. Any opinions, findings, conclusions, and recommendations expressed in this paper are solely those of the authors.

References

[1] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *International Symposium on Microarchitecture*, 1999.

[2] Beyond Semiconductor, "Beyond BA22 Embedded Processor," http://www.beyondsemi.com/25/beyond-ba22-embedded-processor.

[3] E. Biham, Y. Carmeli, and A. Shamir, "Bug attacks," in *Conference on Cryptology: Advances in Cryptology*, 2008.

[4] bjornstar. (2011) nacl_cpuid.c uses vendor string in features check. NativeClient Bug Tracker. Google. [Online]. Available: https://code.google.com/p/nativeclient/issues/detail?id=2508

[5] cbiffle@google.com. (2010) Nacl should accept x86 'int3' instruction or offer a plausible alternative. NativeClient Bug Tracker. Google. [Online]. Available: https://code.google.com/p/nativeclient/issues/detail?id=645

[6] J. Chang, G. A. Reis, and D. I. August, "Automatic Instruction-Level Software-Only Recovery," in *International Conference on Dependable Systems and Networks*, 2006.

[7] K. Constantinides, O. Mutlu, and T. Austin, "Online Design Bug Detection: RTL Analysis, Flexible Mechanisms, and Evaluation," in *International Symposium on Microarchitecture*, 2008.

[8] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "Software-Based Online Detection of Hardware Defects: Mechanisms, Architectural Support, and Evaluation," in *International Symposium on Microarchitecture*, 2007.

[9] M. L. Corliss, E. C. Lewis, and A. Roth, "DISE: a programmable macro engine for customizing applications," in *International Symposium on Computer Architecture*, 2003.

[10] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An Architectural Framework for Software Recovery of Hardware Faults," in *International Symposium on Computer Architecture*, 2010.

[11] T. de Raadt. (2007) Intel Core 2. openbsd-misc mailing list. openbsd-misc mailing list. [Online]. Available: http://marc.info/?l-openbsd-isc&m=118296441702631;

[12] L. Duflot, "CPU Bugs, CPU Backdoors and Consequences on Security," in *European Symposium on Research in Computer Security*, 2008.

[13] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August, "Encore: Low-cost, Fine-grained Transient Fault Recovery," in *International Symposium on Microarchitecture*, 2011.

[14] H. Foster, K. Larsen, and M. Turpin, "Introduction to the new accellera open verification library," 2006.

[15] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Workshop on Workload Characterization*, 2001.

[16] L. C. Heller and M. S. Farrell, "Millicode in an IBM zSeries processor," *IBM Journal of Research and Development*, vol. 48, pp. 425–434, 2004.

[17] M. Hicks, "Practical systems for overcoming processor imperfections," Ph.D. dissertation, University of Illinois Urbana-Champaign, 2013.

[18] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith, "Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically," in *Symposium on Security and Privacy*, 2010.

[19] M. Hicks, C. Sturton, S. T. King, and J. M. Smith. Specs public repository. [Online]. Available: https://github.com/impedimentToProgress/specs

[20] Intel Corporation, "Intel Core 2 Extreme Processor X6800 and Intel Core 2 Duo Desktop Processor E6000 and E4000 Sequence – Specification Update," 2008.

[21] Jennic Limited, "JN5148 Wireless Microcontroller Modules."

[22] Jon Stokes, "Two billion-transistor beasts: POWER7 and Niagara 3," http://arstechnica.com/business/2010/02/two-billion-transistor-beasts-power7-and-niagara-3/.

[23] S. Lemon. (2008) Researcher to Demonstrate Attack Code for Intel Chips. PCWorld. [Online]. Available: http://www.pcworld.com/article/148353/security.html

[24] Advanced Micro Devices, "Revision Guide for AMD Athlon 64 and AMD Opteron Processors," 2005.

[25] ARM, "ARMv4 Instruction Set, Issue C," 1998.

[26] MIPS Technologies, "MIPS R4000PC/SC errata, processor rev. 2.2 and 3.0," 1994.

[27] A. Meixner, M. E. Bauer, and D. Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," in *International Symposium on Microarchitecture*, 2007.

[28] A. Meixner and D. J. Sorin, "Detouring: Translating Software to Circumvent Hard Faults in SimpleCores," in *International Conference on Dependable Systems and Networks*, 2008.

[29] mseaborn@chromium.org. (2009) Check for trailing HLT in x86 is unnecessary. NativeClient Bug Tracker. Google. [Online]. Available: https://code.google.com/p/nativeclient/issues/detail?id=155

[30] mseaborn@chromium.org. (2010) Dynamic loading syscall insists on a trailing HLT on x86-32. NativeClient Bug Tracker. Google. [Online]. Available: https://code.google.com/p/nativeclient/issues/detail?id=585

[31] mseaborn@chromium.org. (2011) Escape from x86-64 inner sandbox using BSF instruction. NativeClient Bug Tracker. Google. [Online]. Available: https://code.google.com/p/nativeclient/issues/detail?id=2010

[32] mseaborn@chromium.org. (2012) x86-64: DATA16 prefix on direct jumps allows sandbox escape on AMD CPUs. NativeClient Bug Tracker. Google. [Online]. Available: https://code.google.com/p/nativeclient/issues/detail?id=2578

[33] S. Narayanasamy, B. Carneal, and B. Calder, "Patching Processor Design Errors," in *International Conference on Computer Design*, 2006.

[34] OpenCores.org, "OpenRISC OR1200 processor," http://opencores.org/or1k/OR1200_OpenRISC_Processor.

[35] G. A. Reis, J. Chang, D. I. August, R. Cohn, and S. S. Mukherjee, "Configurable Transient Fault Detection via Dynamic Binary Translation," in *Workshop on Architectural Reliability*, 2006.

[36] R. Rubenstein, "Open Source MCU core steps in to power third generation chip," 2014, http://www.newelectronics.co.uk/electronics-technology/open-source-mcu-core-steps-in-to-power-third-generation-chip/59110/.

[37] S. R. Sarangi, A. Tiwari, and J. Torrellas, "Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware," in *International Symposium on Microarchitecture*, 2006.

[38] S. Shebs, "GDB tracepoints, redux," in *GCC Developer's Summit*, 2009.

[39] A. L. Shimpi. (2008) AMD's B3 Stepping Phenom Previewed, TLB Hardware Fix Tested. AnandTech. AnandTech. [Online]. Available: http://anadtech.com/show/2477/2

[40] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin, "Ultra Low-Cost Defect Protection for Microprocessor Pipelines," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[41] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery," in *International Symposium on Computer Architecture*, 2002.

[42] Sun, "OpenSPARC T2 Source Code," http://www.opensparc.net/opensparc-t2/download.html.

[43] S. G. Tucker, "Microprogram control for System/360," *IBM Syst. J.*, vol. 6, pp. 222–241, 1967.

[44] I. Wagner and V. Bertacco, "Engineering Trust with Semantic Guardians," in *Conference on Design, Automation and Test in Europe*, 2007.

[45] Xen.org security team. [Xen-announce] Xen Security Advisory 7 (CVE-2012-0217) - PV. [Online]. Available: http://lists.xen.org/archives/html/xen-announce/2012-06/msg00001.html