3-2015

MAGICCARPET: Verified Detection and Recovery for Hardware-based Exploits

Cynthia Sturton

Matthew Hicks

Samuel T. King

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

Recommended Citation

Cynthia Sturton, Matthew Hicks, Samuel T. King, and Jonathan M. Smith, "MAGICCARPET: Verified Detection and Recovery for Hardware-based Exploits", . March 2015.

MS-CIS-15-04

MAGICCARPET: Verified Detection and Recovery for Hardware-based Exploits

Abstract

Abstract—MAGICCARPET is a new approach to defending systems against exploitable processor bugs. MAGICCARPET uses hardware to detect violations of invariants involving security-critical processor state and uses firmware to correctly push software's state past the violations. The invariants are specified at run time. MAGICCARPET focuses on dynamically validating updates to security-critical processor state. In this work, (1) we generate correctness proofs for both MAGICCARPET hardware and firmware; (2) we prove that processor state and events never violate our security invariants at runtime; and (3) we show that MAGICCARPET copes with hardware-based exploits discovered post-fabrication using a combination of verified reconfigurations of invariants in the fabric and verified recoveries via reprogrammable software. We implement MAGICCARPET inside a popular open source processor on an FPGA platform. We evaluate MAGICCARPET using a diverse set of hardware-based attacks based on escaped and exploitable commercial processor bugs. MAGICCARPET is able to detect and recover from all tested attacks with no software run-time overhead in the attack-free case.

Disciplines

Computer Engineering | Computer Sciences

Comments

MS-CIS-15-04

MagicCarpet: Verified Detection and Recovery for Hardware-based Exploits

Cynthia Sturton and Matthew Hicks and Samuel T. King and Jonathan M. Smith

***Abstract*—MagicCarpet is a new approach to defending systems against exploitable processor bugs. MagicCarpet uses hardware to detect violations of invariants involving security-critical processor state and uses firmware to correctly push software's state past the violations. The invariants are specified at run time.**

MagicCarpet focuses on dynamically validating updates to security-critical processor state. In this work, (1) we generate correctness proofs for both MagicCarpet hardware and firmware; (2) we prove that processor state and events never violate our security invariants at run-time; and (3) we show that MagicCarpet copes with hardware-based exploits discovered post-fabrication using a combination of verified reconfigurations of invariants in the fabric and verified recoveries via reprogrammable software.

We implement MagicCarpet inside a popular open source processor on an FPGA platform. We evaluate MagicCarpet using a diverse set of hardware-based attacks based on escaped and exploitable commercial processor bugs. MagicCarpet is able to detect and recover from all tested attacks with no software run-time overhead in the attack-free case.

I. Introduction

Secure systems exactly meet expectations. Software systems, even those intended to be fault-tolerant or secure, trust in hardware to provide certain security properties. In an ideal world we might prove that a processor satisfies all of these properties for all possible traces of execution. An example property that we expand on in the paper is *privilege escalation will not occur*. In the real world, while such a proof is critical, it remains infeasible. Current approaches to verifying hardware include:

(1) Full functional verification: the processor implementation is formally verified against its specification. Formal verification of hardware is a mature field and hardware companies perform extensive verification of hardware modules [14], [56], [45]. However, complete verification of a modern processor remains intractable. Statically verifying that, for example, hardware privilege escalation will never occur is beyond the reach of the state of the art in formal verification.

(2) Testing: extensive test suites are run against the processor. Today, bugs in the hardware that leave it vulnerable [3], [64], [65] still elude such tests. For example, the errata document for Intel's Core 2 Duo processor family [5] contains information on 129 known bugs.

To protect software from exploitable processor bugs left behind by conventional, design time verification we propose a reconfigurable run-time verification system named MagicCarpet. MagicCarpet is a new and effective approach to verifying security properties in hardware. Instead of verifying that a property holds for all executions, MagicCarpet introduces a monitor that detects when the current execution violates the property. The monitor is small, simple, and most important, *verifiable*. Property violations trigger a switch of execution to our small, *verified* simulator to allow the processor to make forward progress securely.

A sketch of the verification approach is to build a provably correct reference monitor and prove that for all possible traces of execution a violation of the security property of interest will be detected, independent of how the violation occurs or what the root cause is; that is, *any trace violating the property will be detected at the point of violation*. Significantly, MagicCarpet also demonstrates a provably correct way to *make forward progress once a violation is detected*.

The benefits of this approach are: (1) we can make guarantees that a security-critical property will not be violated, (2) we can precisely state what guarantees are achieved, and (3) the verification task becomes feasible.

The drawback of this approach is that it can not promise that there is *no* exploitable logic in the processor, but can promise that *any security consequences are limited by the properties being monitored*.

The paper presents five contributions:

- MagicCarpet, the first configurable defense strategy against exploitable processor defects
- An hardware implementation of MagicCarpet and an evaluation that shows MagicCarpet's effectiveness against escaped exploitable bugs from commercial processors
- Automated, formal verification of the invariant monitor hardware
- Automated, formal verification of the recovery firmware
- Automated monitor generation and validation to reduce the threat of misconfigurations

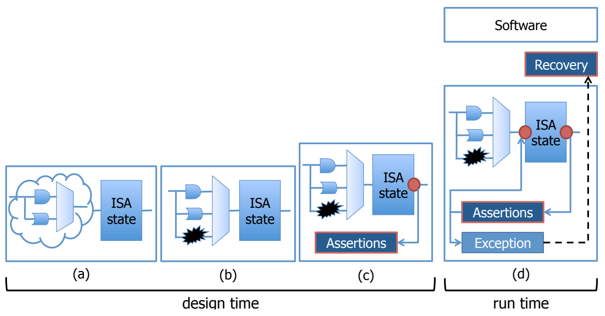

Figure 1: Processor design flow with MagicCarpet: (a) Hardware description language implementation of the instruction set, with *unintentional* bugs. (b) Malicious designer adds *intentional* errors to the processor. (c) MagicCarpet is added to the design as the last action [62], with taps directly on the outputs of ISA state storing elements. (d) MagicCarpet dynamically verifies properties encoded into the fabric, triggering the recovery firmware in the event of a violation.

II. Threat model

A. Lifecycle assumptions

Referring to Figure 1, we assume we are the last ones to touch the processor design, our reference monitor (see Section IV) is inserted into the design and it is not tampered with. The trusted computing base for MagicCarpet includes the specification and verification process and tools, the fabric configuration, and the hardware tools.

B. Architectural assumptions

We assume that the ISA specification is correct [21], as a basis for trust is needed by MagicCarpet. We assume that any hardware modifications or errors exhibit their effects at the ISA level to enable runtime exploitation by the attacker, since the ISA is the processor's interface to software. We require that MagicCarpet has an uncorrupted view of committed ISA-level state. We start from the assumption that it is possible to verify the correctness of an ISA-level state transition using only the current ISA-level state and the instruction. We discuss the limits of this assumption in section VIII. Finally, the focus of our work is the integer core of the processor; we assume the memory hierarchy is correct.

C. Firmware assumptions

MagicCarpet's recovery firmware has six phases (shown in Figure 2): entry; clean-up; fetch; decode; recode and execute; and exit. While we use formal verification to prove some aspects of the firmware, other portions are not amenable to formal verification or are better addressed through functional verification. As described in Section VI-E, we formally verify our recode routines and the instruction decomposition aspects of the decode phase. In addition to what we formally verify, we rely on MagicCarpet invariants to validate the execution of our recoded routines. Therefore, we must trust the entry and exit routines, the clean-up routines, and the unverified parts of the decode logic. We provide details of these assumptions in Section VI-F. This paper focuses on the core of the processor, so errors that can affect the fetch phase of the recovery firmware are out of scope; thus we trust that the simulator correctly fetches the software-level instruction.

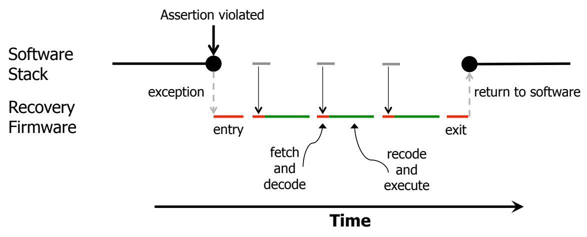

Figure 2: Recovery in MagicCarpet: A property violation invokes the recovery firmware. Recovery is a repeated process of fetching an instruction from software, decoding it, recoding it, and then executing the recoded instruction stream. The recovery firmware returns control back to software when it reroutes execution around the exploitable defect. The green/light portions of the recovery firmware's execution are *formally verified*, while the red/dark portions are not. The fabric monitors the firmware's execution.

D. Attacker model

The attacker is free to take any actions not precluded by our assumptions, either in hardware or in software. This includes an attacker capable of creating and/or exploiting a hardware defect to overcome software protections, such as access controls. An example would be a defect that causes the processor to return from an exception without restoring the privilege level.

Note that exploitable processor bugs are both unintentional [24] and intentional [30], [36]. Our threat model includes both sources and our approach is oblivious to intent as MagicCarpet looks only at effect.

III. MagicCarpet overview

MagicCarpet is a combined hardware/software system for protecting software from exploitable defects in deployed processors. MagicCarpet enforces *security invariants*, which are properties of the ISA necessary to ensure the security of software running on the processor. An example invariant that we will return to is that "*the processor transitions from user mode to supervisor mode if, and only if, there is an interrupt or exception*." Any processor implementation that meets the specification must satisfy this property. Proving that this property holds for an implementation requires a proof across all possible execution traces—currently an intractable task. To sidestep the daunting complexity of the task, we introduce a small, verified monitor that dynamically verifies security properties: MagicCarpet verifies only the current execution. Needing to respond to new threats, we implement the security invariants on a reconfigurable assertion fabric. To handle the rare case of an invariant violation, we provide a verified firmware that attempts to transform software's execution to circumvent the defect. Four key principles guide our design:

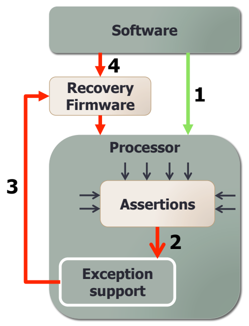

Figure 3: Detection and recovery in MAGICCARPET: 1) Software runs while assertions implemented in *verified* hardware check the processor's state for specification violations; 2) A violation triggers an exception; 3) In-flight contaminated state is flushed and control passes to *verified* recovery firmware; 4) The ISA-level state is backed up, and firmware fetches, decodes, recodes, and executes a software simulation, then loads updated ISA-level state from the simulation into the processor and returns from the exception.

1) *Formally verify added components.* MAGICCARPET should protect against processor vulnerabilities without adding new ones.
2) *Maintain current instruction set architecture (ISA) abstractions.* MAGICCARPET should fix processor imperfections transparently to existing software.
3) *Keep the common case fast.* MAGICCARPET should only add overhead in the rare case that a processor violates one of MAGICCARPET's security invariants.
4) *Make adoption practical.* MAGICCARPET's design and implementation should work well with current industry standards, and should be open source to facilitate adoption.

As shown in Figure 3, MAGICCARPET's hardware monitors security invariants and triggers firmware to recover from violations detected at run-time. MAGICCARPET hardware consists of a set of assertions on architecturally visible states and events. This approach allows MAGICCARPET hardware to track security invariants that are high-level enough to make expressing and reusing security invariants easy, but sufficiently low-level to catch violations effectively and efficiently. MAGICCARPET firmware is a software layer interposed between the processor and the software stack. MAGICCARPET hardware invokes the firmware when it detects a violation. The firmware takes control of the processor, simulating, in software, the necessary ISA-level state transitions. This approach allows the processor to make correct forward progress without affecting upper layers of software.

A. Running example

Security *invariants* are statements of properties of the ISA that must be true of a secure implementation. In the MAGICCARPET design, invariants are dynamically verified by one or more *assertions* over architecturally visible state.

Consider the following component of the example security invariant from Section I: $I_0 \doteq$ *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.

I_0 can be written as a concrete assertion in terms of the ISA-level state in the following way:

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

where `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. The address will always be of the form `0x00000X00` (the "X" indicates a don't-care value).[1]

We break A_0 into three component assertions:

$$A_a \doteq \mathit{assert}(\mathit{risingEdge}(\texttt{SR[SM]}) \rightarrow (\texttt{NPC}[31:12] = 0))$$
$$A_b \doteq \mathit{assert}(\mathit{risingEdge}(\texttt{SR[SM]}) \rightarrow (\texttt{NPC}[7:0] = 0))$$
$$A_c \doteq \mathit{assert}(\mathit{risingEdge}(\texttt{SR[SM]}) \rightarrow (\texttt{reset} = 1))$$

Each of these individual assertions are evaluated at each step of execution and the results are appropriately combined to form a statement that is equivalent to A_0. This example continues in Section IV-A.

B. MAGICCARPET *components and interactions*

MAGICCARPET components and their interactions are shown in Figure 3. Software executes normally on the processor while assertions implemented in hardware check the processor's state for specification violations. The assertions verify that a proposed ISA-level state update is valid given the instruction and the current (already verified) ISA-level state. If the proposed state is valid, no assertions fire, the processor is allowed to commit the proposed state to the ISA level, and execution continues. On the other hand, if an assertion is violated, MAGICCARPET triggers an exception causing any contaminated in-flight state to be flushed, and the recovery firmware to take control.

The recovery firmware backs-up the current architecturally visible state of the processor; these are the registers describing the current state of execution of the software. The firmware performs any additional ISA-level state clean-up which is invariant specific. After the firmware creates a consistent state, it fetches

[1]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 the processor attack attempts to allow user mode execution of supervisor mode pages, MAGICCARPET includes an invariant to detect such misbehavior.

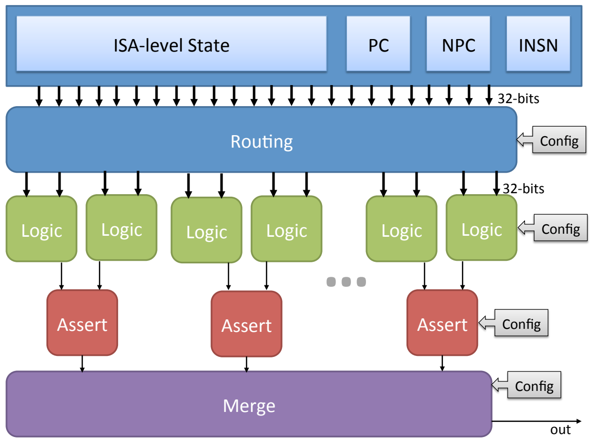

Figure 4: MAGICCARPET fabric: the Routing block sends ISA-level state elements to the Logic blocks; the Logic blocks condense multi-bit state and constant inputs down to a single bit output that is sent to the Assert block; the Assert block compares the previous value of its inputs to the current value, outputing the result as a one bit value to the Merge block; the Merge block combines the Assertion block results to form a higher-level result that indicates if the programmed invariants still hold; this result is tied to the processor's exception generation logic.

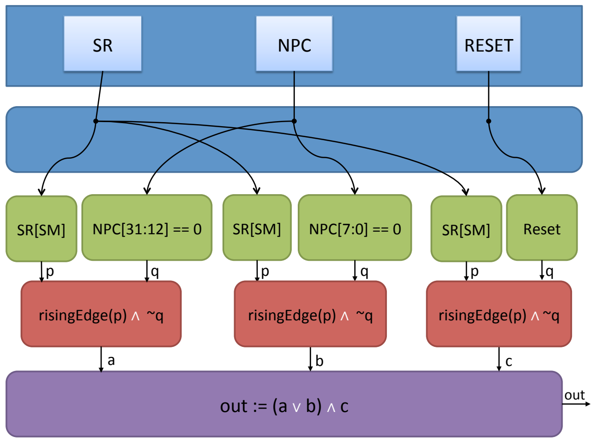

Figure 5: Fabric configured with a macro assertion that detects incorrect privilege escalations.

the instruction pointed to by the software's PC at the time of the assertion violation, then decodes, recodes, and executes the instruction, updating the saved state of the software. This process of fetch, decode, and recode and execute is repeated until execution passes the vulnerable state or transition. At that point, the recovery firmware loads the updated architecturally visible state of the software layer back into the processor and passes control back to the software layer. Figure 2 highlights how MAGICCARPET's recovery mechanism operates. The entire process is transparent to the software layer, which remains unaware of any processor vulnerabilities or protections.

One key aspect of the recovery firmware is that it simulates instructions using a different sequence of instructions to avoid re-triggering the imperfection, similar to BlueChip [33]. For example, MAGICCARPET simulates multiply instructions using shifts and adds. However, MAGICCARPET improves on BlueChip by using proof-based program verification to formally verify the correctness of this recoding. In addition, MAGICCARPET runs as an invisible firmware layer beneath the system software, which is a stark contrast to the BlueChip model of modifying the operating system and enables MAGICCARPET to protect any operating system and application *without* modification.

IV. DESIGN

MAGICCARPET has two main components: the invariant monitor, and the recovery mechanism. The invariant monitor is implemented in hardware and is responsible for detecting violations of ISA state invariants. Should it detect such a violation, it issues an exception that causes control to pass to the recovery mechanism. The recovery mechanism is implemented in software; its role is to allow the processor to make safe forward progress.

A. *Invariant monitor*

Previous work [34] shows that it is possible to use low cost ISA-level assertions as security invariant monitors. In that work, the assertions are "baked-in" to the processor. The problem, as pointed out by the authors, is that baking-in the assertions makes it impossible for users to add new invariants in response to vulnerabilities discovered post-deployment. This limitation makes the approach incomplete given an adversarial threat model. In fact, the authors point out that almost 30% of the processor vulnerabilities escaped the first version of their system. Given this, it is clear that the adversarial threat model of this paper demands a more flexible approach. Thus, in MAGICCARPET, we build a run-time reconfigurable invariant fabric (we call it the configurable assertion fabric) that allows the addition of new assertions to MAGICCARPET post-deployment.

The configurable assertion fabric, depicted in Figure 4, reads in ISA-level state and outputs a signal indicating whether any of the programmed invariants were violated. The configurable assertion fabric is essentially a programmable finite state machine. The configuration data programs the machine with which invariants to check and the ISA-level state acts as the input to the machine. The number of invariants it can monitor concurrently depends on the complexity of the associated component assertions and the number of assertion blocks built into the configurable assertion fabric. The modularity of the configurable assertion fabric makes it possible for us to provide a tool that takes a number of assertions and automatically generates a hardware implementation of the configurable assertion fabric.

Using the example assertions of Section III-A, we now describe each module in the configurable assertion fabric, shown in Figure 5. In our system, we refer to

ISA State	Description	Bits
PC	Program counter.	32
INSN	Current instruction.	32
PPC	PC of last committed instruction.	32
NPC	PC+4.	32
EAR	Effective address register, saved upon exception.	32
SR	Status/Supervision register.	17
ESR	SR at the time of an exception.	17
EPC	PC at the time of an exception.	32
IMMU_SXE	Instruction page is supervisor mode and executable.	1
MMU_UXE	Instruction page is user mode and executable.	1
MMU_EN	Instruction MMU enabled.	1
GPR_WR	General purpose registers are write enabled.	1
GPR_ID	GPR written to.	5
GPR_DATA	Data written to the GPR.	32
DADDR	Address of data coming into CPU.	32
DATA	Data coming into CPU.	32
OPB	Operand B from instruction: `OP` *target*, `OPA, OPB`.	32

Table I: The ISA-level state available to the Routing block.

A_a, A_b, and A_c as component assertions, and A_0 as simply an assertion. The difference being that A_{number} is the implementation of an invariant (a combination of component assertions), where A_{letter} represents a component assertion (corresponding to one assertion block in the configurable assertion fabric).

Routing: The Routing block is responsible for feeding the desired ISA-level state to the Logic blocks. The configuration data determines what state element gets routed to which Logic block. Table I lists the ISA state that is available to the Routing block. To accommodate arbitrary outputs, each Routing block output is 32-bits wide, with zero padding as required. In our running example, `SR[SM]` is output to Logic blocks 0, 2, and 4; `NPC` is output to Logic blocks 1 and 3; and `reset` is output to Logic block 5—as shown in Figure 5.

Logic: Each Logic block implements a configurable comparator operator. Given two inputs A and B, the configuration data can select one of the comparison operators in $\{=, \neq, <, \leq, >, \geq\}$. Additionally, the configuration data can choose to mask off some portion of A or B, or both, or it can substitute a constant value for the value in B. Returning to our running example, Logic block 1 will evaluate $\texttt{NPC\&0xfffff000} = 0$ and output the result; Logic block 3 will evaluate $\texttt{NPC\&0x000000ff} = 0$ and output the result; and Logic block 5 will evaluate $\texttt{reset} = 1$ and output the result. Logic blocks 0, 2, and 4 will evaluate $\texttt{SR[SM]} = 1$ and output the result.

Assert: The Assert block implements component assertions of the form $p \rightarrow q$, possibly across several clocks cycles (*e.g.,* if p is true then 3 cycles later, q is true). If it is ever the case that p is true while q is false, the assertion is triggered and the output of the Assert block will be high. In our example, each of A_a, A_b, and A_c are implemented in their own Assert block. In our fabric, the consequent, q, is always a combinational proposition over ISA state at a single step of execution: it is stateless and is given by the current value sent by the Logic block. However, the antecedent, p, can be a sequential proposition over ISA state: it is potentially stateful, possibly depending on previous values sent from the Logic block. For example, the individual assertions in our example all have the antecedent $\mathit{risingEdge}$ (`SR[SM]`). This proposition is true at time t if and only if `SR[SM]` is low at time $t-1$ and high at time t. The Logic block will output a signal that is high whenever `SR[SM]` is high and the Assert block will determine when a rising edge of `SR[SM]` is seen. Our fabric allows antecedents in one of three forms: $p \in \{\texttt{True}, \neg s_{t-1} \wedge s_t, s_{t-n}\}$. In other words, p can be defined as `True`, in which case the assertion will trigger whenever q is false; or p can be defined to be the rising edge of some ISA state s; or p can be defined to be the value of ISA state s at time $t-n$, where n is also configurable.

The Assert block uses industry standard OVL assertions [29], [6], widely used during testing and simulation of hardware designs. OVL has a variety of constructs, ranging from simple combinational assertions to complex, stateful assertions. The configurable assertion fabric uses only four OVL assertions and our experiments show that these are sufficient to implement all 18 of the invariants used in our evaluation. The assertions are:

- always(expression): expression must always be true
- edge(type, trigger, expression): expression must be true when the trigger goes from 0 to 1 (type = positive)
- 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

Merge: The Merge block takes the outputs from the Assert blocks and combines them as prescribed by the configuration data. The merge block is a configurable truth table. The inputs to the truth table are the Assert block outputs, *e.g.,* the component assertions A_a, A_b, and A_c in our running example. The function defining how the component assertions combine (*i.e.,* the *out* function) is configurable at run time. In our implementation, to reduce scaling issues, the truth table is implemented as a hierarchy of look-up tables. For example, with 16 Assert blocks, rather than a single look-up table with 2^{16} rows, the fabric would have 4 look-up tables with 6 inputs (2^6 rows) each. The outputs of the 3 first-level look-up tables make up the input to a second-level look-up table, the output of which is the output of the Merge block.

We can now complete our running example. Let err_a be the output of the Assert block for A_a, and let err_b and err_c be the output of the Assert blocks for A_b and A_c, respectively. Remembering that the output of each Assert block will be high when the assert triggers, *i.e.,* when the invariant is violated, we combine the results of the component assertions in the following way.

$$err_0 = (err_a | err_b) \& err_c$$

As desired, err_0 will be high whenever A_0 is false, *i.e.,* whenever the A_0 assertion is triggered.

Configuration data: The configuration data is provided by trusted, low level, software (e.g., the system BIOS) at initialization (originally, we imagine configuration coming from processor or motherboard manufacturers). It is the mechanism by which the configurable assertion fabric is configured and portions of the configuration data are fed into each block at the appropriate stage. Built in to the design of each block is a check that the incoming configuration data is well-formed. Badly formed configuration data results in disabling the entire configurable assertion fabric. In this case, the output of Merge will always be low—the processor is unprotected, but usable. To avoid misconfigurations, we show in Section V that it is possible to use formal methods to validate configuration against common mistakes.

V. Verification

A. Configurable Assertion Fabric

We used Cadence SMV [42] for the verification of the configurable assertion fabric. Cadence SMV is a BDD-based [13] symbolic model checker for finite state systems. It can operate directly on Verilog, which means that our verification was done directly on the synthesizable RTL (register transfer level) description of our configurable assertion fabric, rather than on a model of the fabric. For each component of the reference monitor shown in Figure 4, we verified that the implementation meets the specification. The Verilog source code for each module, the formal specifications, and the linear temporal logic (LTL) formulas we used for verification are available on our website [1].

The first step of the verification effort was to formalize a complete specification of each component. The components are relatively simple, and in most cases, formally specifying their behavior involved little more than extracting the information from the design documents. However, in a couple of cases, the effort of formalizing the specification brought out ambiguities in the design and it was necessary to revisit the design phase of the process. We discuss two such instances here.

The Routing block is implemented as a series of MUXes. For each, the input lines are the available components of ISA state and the $select$ line is driven by configuration data. Although the synthesized configurable assertion fabric has a fixed number of input lines to each MUX, in the Verilog source code the number of input lines and the width of the $select$ signal are given as parameters. It is possible for $select$ to have values which do not map to any available input line to the MUX. For example, if the MUX had 100 32-bit input lines, $select$ would need to be 7 bits wide, but some values of $select$ (values 100-127) would not map to any valid ISA state component. In the English-language design document the maximum value of $select$ was given as an integer that exactly matched the number of input signals to the MUX; it was not possible for $select$ to ever have a value that was out of range for the MUX. This undefined behavior quickly became apparent when we tried to formalize the specification. As a result, we went back to the design phase and introduced the $configInvalid$ signal. The $configInvalid$ signal gets propagated through each of the components in the configurable assertion fabric. Each component can set the $configInvalid$ signal if necessary, but can never clear it. In the case of the Routing block, $configInvalid$ is set whenever $select$ has an invalid value.

A second design decision that came out of our specification effort was the realization that configuration data should be stable. That is, the configuration for the configurable assertion fabric should not be able to change on each clock step. Rather, when MagicCarpet is enabled, the configuration data should be frozen. This was implicitly assumed by the design, but we had to make the assumption explicit during the verification. Currently, our system does not prevent configuration data from changing while MagicCarpet is active, we expect that adding that constraint to the system will not be difficult.

Routing and Logic: The Routing and Logic blocks contain only combinational logic: the output depends only on the current inputs. For each module, we verified its correctness by specifying the correct output for all possible combinations of input, and then proving that the implementation meets the specification.

Assert: Unlike the previous two blocks, Assert contains sequential logic, and therefore, internal state. Recall that the Assert block implements assertions of the form $p \rightarrow q$, where p is in one of three forms: *assert always* ($p = \texttt{True}$), *assert on rising edge* ($p = \neg s_{t-1} \wedge s_t$), and *assert after n clocks* ($p = s_{t-n}$). A $select$ input is used to choose which form p will take; it is provided by the configuration data. The signals s and q are each driven by the out signal of a Logic block.

For the verification of the Assert block, we case split on the $select$ input and separately verify the behavior of the block for each possible value of $select$. During the course of verification, we found an error in the $configInvalid$ signal. A logical AND was used where an OR was needed. This was the only implementation error we found during verification.

Merge: The Merge block calls for software formal verification as opposed to hardware formal verification. It is implemented as a look-up table and the configuration data provides the contents of the table. The behavior of the block is entirely determined by the values in

$$
\begin{aligned}
\langle\text{configuration}\rangle &\models \langle\text{assertOp}\rangle\ \{\langle\text{binaryOp}\rangle\ \langle\text{assertOp}\rangle\}\ \textbf{;}\\
\langle\text{assertOp}\rangle &\models [\langle\text{unaryOp}\rangle]\langle\text{assert}\rangle\\
\langle\text{unaryOp}\rangle &\models \textbf{!}\\
\langle\text{binaryOp}\rangle &\models \textbf{AND}\ \mid\ \textbf{OR}\\
\langle\text{assert}\rangle &\models \langle\text{always}\rangle\ \mid\ \langle\text{edge}\rangle\ \mid\ \langle\text{next}\rangle\ \mid\ \langle\text{delta}\rangle\\
\langle\text{always}\rangle &\models \textbf{ALWAYS (}\ \langle\text{logicBlock}\rangle\ \textbf{)}\\
\langle\text{edge}\rangle &\models \textbf{EDGE (}\ \langle\text{logicBlock}\rangle\ \textbf{,}\ \langle\text{logicBlock}\rangle\ \textbf{)}\\
\langle\text{next}\rangle &\models \textbf{NEXT (}\ \langle\text{INTEGER}\rangle\ \textbf{,}\ \langle\text{logicBlock}\rangle\ \textbf{,}\ \langle\text{logicBlock}\rangle\ \textbf{)}\\
\langle\text{delta}\rangle &\models \textbf{DELTA (}\ \langle\text{INTEGER}\rangle\ \textbf{,}\ \langle\text{INTEGER}\rangle\ \textbf{,}\ \langle\text{isaState}\rangle\ \textbf{)}\\
\langle\text{logicBlock}\rangle &\models \langle\text{masked}\rangle\ \langle\text{compareOp}\rangle\ (\langle\text{masked}\rangle\ \mid\ \langle\text{INTEGER}\rangle)\\
\langle\text{masked}\rangle &\models \langle\text{isaState}\rangle\ [\textbf{AND}\ \langle\text{INTEGER}\rangle]\\
\langle\text{compareOp}\rangle &\models \textbf{EQ}\ \mid\ \textbf{NEQ}\ \mid\ \textbf{GT}\ \mid\ \textbf{GTE}\ \mid\ \textbf{LT}\ \mid\ \textbf{LTE}\\
\langle\text{isaState}\rangle &\models \textbf{see Table I}
\end{aligned}
$$

Figure 6: E-BNF grammar supported by the reprogrammable invariant fabric.

the table. This means that verifying the Merge block requires validating the configuration data, which we cover next.

Discussion: Ultimately, the fabric's behavior is determined by the configuration data and it is up to the the processor/motherboard manufacturer to provide a correct configuration. A misconfigured fabric could fail to provide the intended protections. We show that it is possible to use formal verification to guard against misconfigurations in three ways. First, we protect against invalid configurations that would result in unpredictable results. We verify that if any of the individual components report an invalid configuration, then the composed fabric will not fire any assertion failures. This behavior represents a trade-off in the design space. On the one hand, an accidentally misconfigured fabric, which will never trigger an assertion, is not protecting the user as the user probably expects. On the other hand, never firing in the presence of misconfigured data has the benefit of being a stable behavior—it is what exists today. An alternative is to always fire when the fabric is misconfigured, but this would give an attacker an avenue for launching a denial of service attack—making MagicCarpet a new avenue of attack, something we go to great lengths to avoid.

Second, we built a software tool to generate the configuration data from higher-level assertion statements. Although only prototypical, we hope that further developing this tool will make generating correct configuration data relatively easy for the user. Figure 6 shows the grammar the the tool implements.

Third, we built a validation tool that performs the following sanity checks on the configuration data:

- Are any of the assertions unsatisfiable?
- Are any assertions tautologically violated?
- Are there zero assertion checks configured?
- Does the configuration imply that an assertion check will fail at every step of execution?

If any of these tests come back positive, a misconfiguration error is reported along with information about the offending assertion(s). The user can run this tool before loading the configuration data into MagicCarpet. We used the z3 SMT solver [23] as the back end to this tool. We tested its performance over randomly generated configuration data for varying numbers of assertion blocks. Figure 7 shows the results of that experiment. The saw-tooth pattern is a result of the hierarchical lookup tables used to implement the merge block. As the table in the last level fills up, more calls to the SMT solver are needed and the total validation time increases. Once a new level in the hierarchy is introduced, the table in the last level is mostly empty and, while each call to the solver takes longer, there are many fewer calls for a net gain in time. It is necessary to note that while we formally verify the functional correctness of each module in the configurable assertion fabric, we manually audit the connection between modules. That is, we manually check that every module's output signals are appropriately tied to the next module's input signals. There is no logic involved in the composition and our naming convention made the checks straightforward. Our end-to-end verification of the invalid configuration signals, mentioned above, does *not* rely on this manual audit.

VI. Recovery firmware

When the invariant monitor detects that the processor has violated one of the monitored invariants, an exception is triggered and control passes to the recovery firmware. The firmware is responsible for helping software safely execute sections of code that violate a security-critical processor invariant. MagicCarpet's

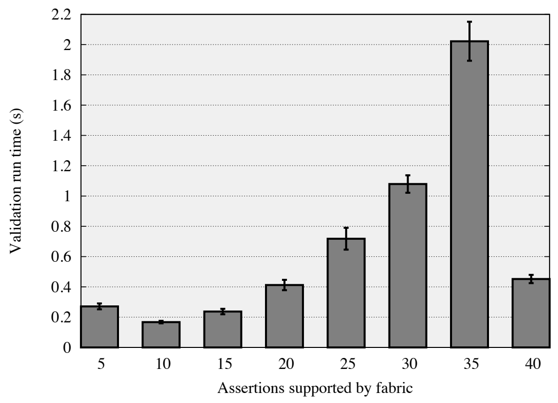

Figure 7: Running time of the configuration validation tool for randomly generated configuration data of consisting of varying number of assertions. For each fabric size, we run the configuration validation tool on 30 randomly generated configurations. The mean runtime of validating the 30 sets along with the 95% confidence interval.

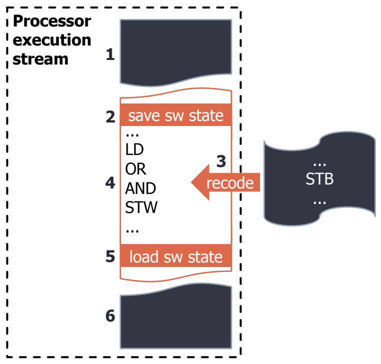

Figure 8: MagicCarpet's recovery mechanism: 1) Software executing normally on the processor 2) When the configurable assertion fabric detects a processor invariant violation, control passes to the recovery firmware which backs-up the software's state and exception registers. 3) The recovery firmware fetches, decodes, recodes, and 4) executes several instructions from the software level. 5) Once past the error, the recovery firmware loads the processor with the software state in the simulator and passes control back to the software level. 6) Software resumes normal execution

recovery mechanism repurposes the processor's built-in exception handling logic as a checkpointing and rollback mechanism, avoiding the complex and heavy-weight hardware structures required by other proposals [55]. A key observation is that the processor exception handling mechanism already provides a version of checkpointing and rollback, just with a checkpoint window spanning from the fetch stage to right before the instruction's changes are committed to the ISA-level state (*e.g.,* three stages on the traditional MIPS 5-stage processor). When a MagicCarpet exception occurs, the processor flushes in-flight state and passes control to MagicCarpet's recovery firmware.

The recovery firmware pushes software state forward by simulating processor execution of the troublesome code. It does so by translating each instruction to an equivalent sequence of instructions that does not include the original instruction, *i.e.,* a subset of the original ISA. By changing the set of available instructions, the firmware mutates the original instruction stream to one that achieves the same ending ISA-level state, but which takes a different path, with different intermediate states. This protects against reactivating both instruction- and state-sensitive processor defects. Figure 2 provides an overview of how recovery works. The mechanism consists of entry and exit routines, state clean-up routines, routines that fetch and decode instructions from software, and recoding routines. We discuss each of these components below.

Figure 8 provides an overview of how the recovery process works to push software execution forward, around processor bugs by recoding the original instruction stream and executing the recode stream on the processor. MagicCarpet's recovery firmware consists of entry and exit routines, routines that fetch and decode instructions from software, and recoding routines.

A. *Hardware/firmware interface*

One of MagicCarpet's goals is to not increase the complexity of the processor, but some added complexity is required to support software-level recovery. It is important for correct recovery that the firmware know the source of the invariant violation. This requires the addition of an ISA-level register that the firmware can read to direct state clean-up and recoding. Another issue is that invariant violations can occur even in software that is executing with interrupts disabled. This means that invoking the firmware could overwrite special purpose registers not backed-up by software. To solve this issue, MagicCarpet adds ghost versions of all exception registers and creates an additional ghost register to be used as a temporary holding place for the recovery firmware's entry routine—we usually put software's stack pointer there. Without this extended interface between hardware and the firmware, it is impossible to invoke the recovery firmware without losing some crucial state.

Another addition to the hardware/firmware interface is a set of registers that allow for loading configuration data into the fabric. There are three 32-bit registers: one for data, another for address, and a final register for alerting the fabric that the address and data are valid (called strobe).

B. *Firmware/software interface*

When the processor issues a bug detection exception, MagicCarpet's recovery firmware is tasked with pushing the state of the software layer forward, so that it can continue executing on its own, past the invariant violating code. After the recovery firmware saves the current state of the software layer (as stored in the processor), initializes itself, and determines what invariant was violated, it is ready to assist the software layer. This requires that the firmware first clean-up any contaminated state not flushed by the exception. As explained in Section VII, there are three clean-up methods (of different overhead) that depend on the invariant violated. Once the state is consistent, the firmware starts by reaching into the address space of the software layer to read instructions and data values. The recovery firmware does this by manually walking the TLBs (if the MMU was enabled at the time of detection) and fetching the required value straight from memory.

MagicCarpet firmware then simulates the interrupted software using a different set of instructions to avoid re-exercising the exploitable processor logic. Although our simulation is general purpose and our technique of simulating instructions using a different set of instructions enables MagicCarpet to avoid a wide range of defects, our hardware/firmware interface, by exposing which invariant was violated allows for ad hoc recovery routines. The custom recovery routines are more efficient than our general purpose simulator, but the simulator is at least a reliable backstop.

Entry and Exit: When the monitor detects a processor invariant violation and triggers an exception, control passes to the recovery firmware's entry routine. The entry routine starts by saving the current state of software, *i.e.,* general-purpose registers, the exception registers, and stack pointer. MagicCarpet includes special purpose, ISA-level registers for this purpose. Once the software's state has been saved and the state cleaned, simulation begins at the instruction pointed to by the saved PC. This is the instruction that software was executing when the violation occurred. The new registers are protected by MagicCarpet hardware: access is allowed only when the current instruction pointer is within a specified, hardcoded memory range—the memory used by MagicCarpet firmware and made unavailable to the OS or other software by the hardware.

After simulating all of the instructions that were in the processor pipeline at the time of the invariant violation, the firmware restores the software-level state back into the processor and returns control of the processor to the software layer. Because an attempt at recovery may itself violate an invariant, the recovery firmware is designed to handle recursive invariant violations, although only if they occur outside of the entry and exit sequences (Section VI-C).

Fetch and Decode: Starting from the last committed instruction, the recovery firmware fetches and decodes, on an instruction-by-instruction basis, instructions from the software layer. Doing so requires reaching into the address space of the software layer to read instructions and data values. If necessary, the firmware manually walks the TLB of the software layer before fetching the required value from memory.

The decode process works by breaking an instruction into meaningful chunks as dictated by the instruction set specification, *e.g.,* opcode, operands, and constant. Once the opcode is known, the simulator calls the associated instruction recoding routine.

Recode and Execute: The recode routine simulates the interrupted software using a different set of machine instructions. This avoids re-exercising the exploitable processor logic. An example recoding used in MagicCarpet's recovery firmware is replacing a multiply instruction with a series of shifts and adds. To avoid adding bugs to the system by incorrectly recoding instructions, we formally verify the equivalence of the recoded instruction stream to the software-level instructions that they replace—detailed in Section VI-E.

C. Handling recursive violations

Recovery in MagicCarpet is a trial and error process. Because an attempt at recovery may itself violate an invariant, MagicCarpet is designed to handle recursive violations—outside of the entry and exit sequences. This allows the recovery firmware to keep changing the sub-ISA used to recode the original instruction stream, increasing the odds of recovery. The ability to handle recursive violations also allows recovery firmware designers to sacrifice likelihood of recovery in favor of lower average run time overheads; since they can also try the safer recovery method if the faster method fails. Experimental results in Section VII highlight the difference in speed of two different recovery approaches.

The first obstacle to handling recursive invariant violations is protecting the ISA-level state that taking and exception atomically updates. MagicCarpet handles this by creating a special stack for all atomically updated registers. The recovery firmware stores a pointer to the most recent frame at a known address in it memory space. Since the frames are all the same size, traversing between the frames of different recovery firmware invocations is trivial. All atomically updated registers and the pointer are saved/updated in the entry routine and the most recent frame is unloaded into their associated registers in the processor and the pointer decremented in the exit routine.

The second obstacle is that the recovery firmware cannot simulate itself. This creates problems when invariant violations occur while the recovery firmware is fetching or decoding instructions from software, *i.e.,* not actually recoding or executing the recoded instruction stream. In these cases, which the recovery firmware knows by inspecting the address associated with the violation, recovery will just restart in hopes that the act of taking the exception was enough of a disturbance to avoid the violation. To avoid live locks due to this restriction, we avoided complex control paths in the fetch and decode code to make its execution pattern very regular. None of the attacks used in our evaluation threatened the fetch or decode parts of the recovery firmware. If they did, we could easily recode the recovery firmware to work around the vulnerability. If the recovery firmware is recoding or executing the recoded instruction stream, it will restart the simulation process, but with a different recoding algorithm.

D. Software managed reliability

Results from experiments (Section VII), where MagicCarpet was tasked with monitoring an increasing number of bugs, expose the problem of false bug detections. Bug detector resources are finite (Section VII), therefore, when MagicCarpet needs to monitor for activations of multiple bugs, there is a chance that the bug signatures will involved the same state values. To prevent contamination due to missed bug activations, MagicCarpet must combine competing bug signatures in a pessimistic way, creating the potential for false positives.

Dynamically enabling and disabling bug detectors can reduce false detections while maintaining a consistent ISA-level state. Since bug signatures in MagicCarpet are dynamically reconfigurable, both hardware and software have an opportunity to manage which bugs detectors are active. In the case of software, MagicCarpet extends the processor's interface with software, allowing software to manage its own reliability and run time overheads by controlling which processor bugs it's exposed to and when. Experimental results validate the power of this mechanism to reduce false detections, increasing software performance.

E. Proving correctness

Because MagicCarpet firmware simulates instructions *without* using the instruction it is simulating, the recovery firmware is more complex than a traditional instruction-by-instruction simulator. For example, we simulate add instructions using a series of bit-wise Boolean operations, shifts, and comparisons to check for carry and overflow states. These bit-manipulation operations are difficult to reason about manually, which motivates our use of formal methods to prove the correctness of our implementation.

The code that we prove falls into three categories. First, we prove correct helper functions that we compose together to simulate instructions. Examples include sign extend, zero extend, and overflow and carry logic for arithmetic operations. For each function we formalized its specification and verified the implementation against the specification. Second, we prove correct simulation of the ISA instructions. For example, we prove that a series of shifts and adds are equivalent to a multiply instruction. The alternative implementation never uses the simulated instruction. Third, we specify and prove correct the first phase of the decode logic. We prove that during simulation, the decode logic correctly slices an instruction into its component parts (*e.g.,* opcode, mode, and operand). We do not verify the mapping from opcode to alternative implementation. For example, we do not verify that a multiply instruction from the software layer always invokes the multiply recoding routine in our simulator. All of our source code, with our specifications, can be found on our web site [1].

To verify our implementation, we used VCC [19]. VCC enables sound verification of functional properties of low-level C code. It provides ways to define annotations of C code describing function contract specification. Given the contracts, VCC performs a static analysis, in which each function is verified in isolation, where function calls are interpreted by in-lining their contract specifications. Our verification efforts, which took place after all conventional software tests were passed, revealed subtle bugs in our sign extend helper function and our divide instruction implementation that were caused by implicit, yet incorrect, assumptions we made when writing the simulator (mostly to do with signed vs unsigned numbers). These implicit assumptions were revealed as bugs when formal verification made them explicit.

The results of our verification effort is approximately 3000 lines of formal specification—very similar to the number of lines of code. Verification with VCC is quick, on the order of minutes. Even with tight feedback loops between firmware developer and the tester, formal verification exposed three bugs in our code. Interestingly, the processor bug from the processor we use that has eluded several attempts at patching [47] is related to the same signed vs unsigned issues that caused the bugs reported by VCC in our simulator implementation.

F. Assumptions

Here we elaborate on the assumptions stated in Section II-C. The firmware makes three assumptions:

1) The processor is in a state that can be interrupted by the exception generated by an invariant violation.
2) The recovery firmware's entry and exit routines execute without violating an invariant. To allow for handling of recursive invariant violations, the first and last action the recovery firmware performs is backing-up to and restoring from memory the state overwritten by the processor when it handles an exception (*e.g.,* the Status Register).
3) The recovery firmware correctly decodes the software-level instruction. While we do prove that the firmware correctly decomposes instructions into its constituent parts, *e.g.,* opcode and immediate, we do not prove that, given the opcode, the simulator correctly executes the appropriate recoding routine.

G. Approach weaknesses

Complete recovery is not always possible, even if all the assumptions in Section II-C hold. Generally, if there is insufficient vulnerability-free redundancy available or the recoding routines do a poor job at exploiting the vulnerability-free redundancy, recovery will fail. One concrete example for our implementation is a defect involving a write or read from the byte-bus (a low bandwidth peripheral bus used by our processor). As its name implies, the byte-bus is an 8-bit wide peripheral bus that can only be accessed at the byte level. If there is a flaw in the processor involving byte-bus accesses, there is no way for MagicCarpet firmware to recode execution around the bug.

Possible approaches to handling this type of failure includes simply updating the recovery firmware or even extending the ISA further to allow MagicCarpet to communicate cases of incomplete recovery to the software level. Another option is for processor designers to ensure that functionality that has the least redundancy is verified the most.

VII. MagicCarpet evaluation

This section describes a FPGA-based implementation of MagicCarpet and how we test that implementation with an array of attacks based on exploitable bugs found in commercial processors. This section also details the four different recovery strategies required to push software past all the attacks in our test bed. Lastly, this section looks at hardware overheads due to different sized configurable assertion fabrics and how MagicCarpet scales to more complex processors.

A. Implementation

To evaluate the performance and efficacy of MagicCarpet we implement it inside the OR1200 [46] processor. The OR1200 is an open source, 32-bit RISC processor with a 5-stage pipeline, separate data and instruction caches, and MMU support for virtual memory. It is popular as a research prototype and has been used in industry as well [51]. The processor is representative of what you would see in a mid-range phone today.

For the invariant fabric, we build a program that automatically generates the fabric hardware when given the number of Assert blocks that the fabric must support. Besides making it easy to explore the impact of tuning different fabric parameters, using a tool to generate the fabric creates a regular naming and connection pattern that allows us to verify the structural connections of arbitrary fabrics using an induction type approach.

For a complete system, capable of booting Linux, we implement the processor and fabric combination as the heart of a system-on-chip that includes DD2 memory, an Ethernet controller, and a UART controller. We implement the system-on-chip on the FPGA that comes with the Xilinx XUP-V5 development board [25]. We conservatively clock the system at 50 MHz. Section VII-D explores the hardware overheads required by different configurations of MagicCarpet.

B. Effectiveness

To study how well MagicCarpet protects a vulnerable system against attack, we weakened the OR1200 by introducing the 14 vulnerabilities listed in Table II.

ID	Synopsis	Detect	Recover	Strategy
1	Privilege escalation by direct access	✓	✓	C
2	Privilege escalation by exception	✓	✓	C
3	Privilege anti-de-escalation	✓	✓	C
4	Register target redirection	✓	✓	C
5	Register source redirection	✓	✓	S
7	Disable interrupts by SR contamination	✓	✓	S
10	EPCR contamination on exception exit (to PC)	✓	✓	S
13	Register source redirection	✓	✓	S
8	EEAR contamination	✓	✓	C+S
9	EPCR contamination on exception entry (from PC)	✓	✓	C+S
6	ROP by early kernel exit	✓	✓	B+S
11	Code injection into kernel	✓	✓	B+S
12	Selective function skip	✓	✓	B+S
14	Disable interrupts via micro arch	✓	✓	B+S

Table II: Attacks based on previously discovered exploitable processor errata from AMD processors [34], ordered by increasing recovery complexity. For each, can MagicCarpet detect it (**Detect**)?, correct any ISA state contamination and push software execution around it (**Recover**)?, and recovery strategy (**Strategy**: C - correction; S - Simulate; B - backtrack) used.

The vulnerabilities come from previous research that identified exploitable processor errata from several years of AMD processor errata [4], [34]. These errata capture the types of exploitable processor bugs that escape current verification methodologies. To map each erratum into an attack, we take its effect and connect it as the payload to a variety of stealthy triggers.

The previous paper that proposed the attacks also built them directly into the processor. Our goal is to show that the MagicCarpet is expressive enough to allow us to implement all the same invariants on the configurable assertion fabric. To this end, we configured the fabric with the 18 assertions described in Tables III and IV.

For each attack, we write a program for the OR1200 that triggers the attack and reports if the attack was successful. Table II provides the results of this experiment: MagicCarpet is expressive enough to implement all 18 invariants, thus it can detect each attack.

C. Recovering from the attacks

Detecting attempted attacks is only part of a complete system—otherwise every attack becomes a denial-of-service attack. The other half of the challenge is maintaining an ISA-level state consistent with the specification and helping software to continue to execute safely in spite of processor imperfections. To show that MagicCarpet meets this challenge, we verify that software makes forward progress and that the state of software is consistent with the instruction set specification. The column labeled "Recover" in Table II shows, for each attack, if MagicCarpet's recovery firmware is able to undo any contamination and then ensure software's forward progress.

The result of this experiment is that we are able to recover from all attacks. This experiment also reveals that there are four possible recovery strategies. The lowest overhead recovery is "Correction," which involves resetting the contaminated ISA-level state to a safe value (*e.g.,* setting the processor mode to user mode

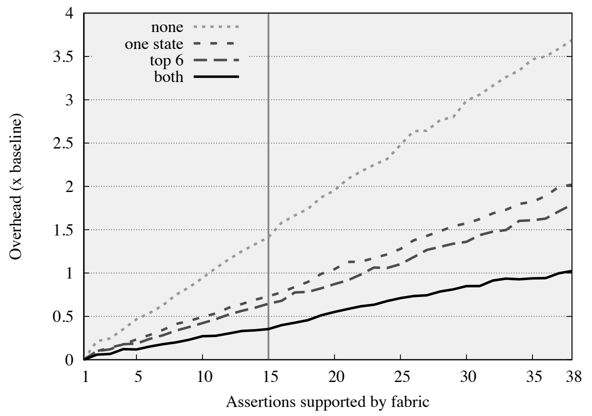

Figure 9: Hardware overhead with respect to the number of assertions supported by the configurable assertion fabric, evaluated at four optimization levels. The vertical line represents the number of Assert blocks required to implement the seven assertions needed to detect all of our attacks. 38 is the number of Assert blocks to implement all invariants.

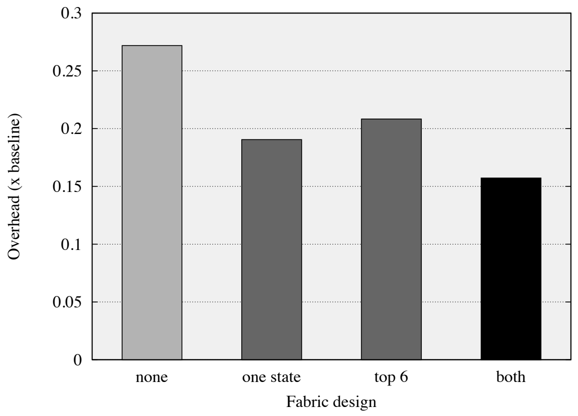

Figure 10: Timing delay overhead, averaged across all fabric sizes, with respect to the number of assertions supported by the configurable assertion fabric at four optimization levels.

if the privilege escalation macro assertion fires) and then continuing execution. The next level of recovery is "Simulate." "Simulate" occurs when there is no contaminated state to correct, but MAGICCARPET must help software to make forward progress by recoding the instruction stream to bypass the attack. The simulator introduces large increases in overhead compared to "Correction"-level recovery. "Simulate with backtrack" is the third recovery strategy. It is similar to "Simulate," but involves starting simulation from an earlier (already executed) instruction. This strategy is useful for control flow oriented violations. The final and most costly recovery strategy is "Simulate with correction." This is needed when an attack contaminates a value that cannot be recalculated simply by backtracking. In this case the recovery firmware must recalculate the value itself. While able to recover contaminated values in each of our experiments, it is possible that full clean-up may not be possible. In this case, the recovery firmware can simulate at process terminating event to kill the process, *e.g.,* seg. fault.

D. Evaluating the cost of MAGICCARPET

Experiments with a wide range of attacks demonstrate that MAGICCARPET is effective at both detecting and recovering from exploitable processor defects. But what is the cost of MAGICCARPET protections? Here, we evaluate the cost of MAGICCARPET in terms of the hardware overheads of the reconfigurable invariant fabric and the software run time overheads due to the recovery firmware. As a reference point, previous work on deployed bug patching (see Section IX) entails hardware overheads of up to 200% and run time overheads of up to 100% in the *common* case.

Processor designers can select the number of assertions supported by MAGICCARPET's invariant fabric: they can tradeoff assertion support for hardware overhead. To show the hardware overheads associated with different sized fabrics, we used our fabric generation tool to build fabrics with support for as little as 1 assertion to as many as 38 assertions (the number required to implement all the invariants in Table IV).

Figure 9 shows how the hardware area overhead changes as the number of assertions supported by the configurable assertion fabric increases. Similarly, Figure 10 shows how the hardware delay (determines the maximum clock frequency) overhead changes as the number of assertions supported by the configurable assertion fabric increases. Each figures contains data at four points in the fabric design space: (none) no optimization, the fabric favors expressibilty over overheads. (one state) previous work shows that 83% of invariants targeted at protecting against security-critical processor bugs only use the first of two state inputs of the Logic block. Thus, "one state" represents using Logic blocks with only one state input. This optimization also reduces the number of required Routing blocks by 50%. (top 6) is an optimization on the Routing block that leverages the observation that 76% of invariants require the same six ISA-level state items. Thus, "top 6" represents replacing the Routing blocks with new Routing blocks that only handle the six most frequently used state elements. Finally, (both) represents a fabric with both of the aforementioned optimizations.

To identify opportunities for future optimization we explore the relative composition of each fabric block to the overall fabric. Figure 11 shows, for each optimization level, how much each block type contributes to overall hardware area of the fabric. This experiment shocked us: we ran this experiment after applying the "top 6" optimization, but before we thought that we needed the "one state" optimization. Before this experiment, we were under the impression that the crossbar switch nature of the Routing block and the fact that there are four Routing blocks for each Assertion

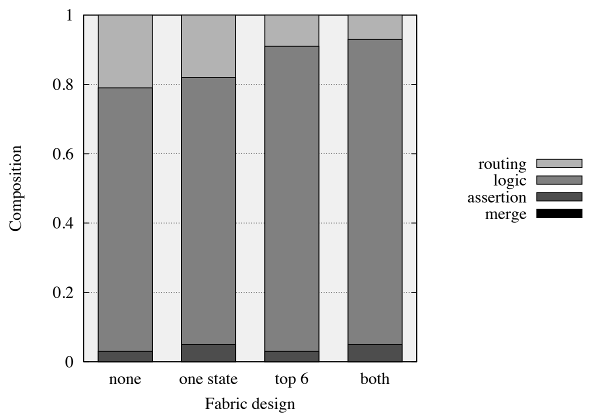

Figure 11: Proportion of fabric area each fabric component contributes. Note that these results do not capture wiring overhead, to which the Routing block is the dominate contributor.

block make the Routing block the main contributor to overhead. Figure 11 clearly shows that the Logic block is the main contributor; hence, we design the "one state" optimization.

There is still plenty of opportunity for overhead reductions, but that level of engineering effort is out of scope for this paper.

E. Generalizing the results

Building a hardware prototype of MagicCarpet is essential both as a way to expose implicit assumptions and as a way to discover traits of our approach (*e.g.,* that the Logic block is the largest component of the fabric's hardware overhead; we thought that it was the Routing block). For the hardware prototype, we chose the OR1200 because it is a mature and open source processor implementation on which we could build and evaluate our MagicCarpet design. The choice to focus on building the system in real hardware leaves unanswered the question of how our approach scales to the x86 class of processors.

We expect that the MagicCarpet design will scale well. Complexity in commercial processors largely comes from optimizations meant to increase performance. These optimizations add state below the level of the software interface, *i.e.,* without changing the architecturally visible state. MagicCarpet, on the other hand, monitors only the architecturally visible state (for example, the CPU registers and general purpose registers). This means that it is possible that while MagicCarpet would be more complex on more complex processors, its overhead relative to the rest of the processor would be less than our prototype.

While we expect the hardware aspects of MagicCarpet to scale well, the effort required to build and verify the decode and recode logic in MagicCarpet's firmware would grow considerably for commercial processors—roughly linear with the number of instructions in the ISA. To concretize this scaling, it took two graduate students two weeks to build and verify the recovery firmware for the OR1200. Given that there is almost a two-order magnitude increase in number of instructions from OR1200 to x86, we expect that the effort required for a commercial processor would be 400 person-weeks—less than a year for a 10 person team: reasonable for a large company. Additionally, being at the ISA level reduces the changes required when porting MagicCarpet between processor families.

VIII. Limitations

MagicCarpet focuses on protecting security-critical processor state, it does *not* attempt to address processor exploits that target general-purpose state, exploits that target the memory hierarchy, or to prevent side channel emanations. MagicCarpet relies on the observation that there is a security-critical portion of processor state and that the processor updates this state in a few, simple ways. General-purpose processor state contradicts this: there is a plethora of general-purpose state and it is updated in a myriad of ways. To apply MagicCarpet to general-purpose functionality, the increase in state would cause the routing block to become more complex and the myriad of ways to update general-purpose state would cause a dramatic increase in logic block complexity. Additionally, it is unclear if including general-purpose state would require MagicCarpet to support more types of OVL assertions. From a more pragmatic level, adding such complexity and overhead to the processor comes with questionable benefit—possibly making formal verification of MagicCarpet intractable: research shows that software can protect itself against general-purpose processor imperfections with little run-time overhead and no added hardware complexity [16].

Note that it is possible, especially in the case of malicious processor imperfections, that a contamination of a general-purpose register can make its way—with the help of unwitting privileged software—into ISA-level state protected by MagicCarpet. In these situations, software can leverage its flexibility advantage over hardware and patch itself so that it doesn't trigger the malicious imperfection or so that it doesn't blindly carry potentially malicious values into protected state. It is also possible to use previous research in a targeted manner to verify the computation of values that will eventually be stored in protected state.

Another limitation of the MagicCarpet's monitor hardware is the fabric configuration data. While we do show that it is possible and low overhead to validate aspects of the configuration using formal methods (*e.g.,* the configuration does not cause constant invariant violations), we do not prove that a configuration does what the designer intends it to do. To reduce the threat of misconfigurations, we designed and formally verified that the fabric blocks (*i.e.,* routing, logic, assertion, and merge) fail gracefully when misconfigured—essentially disabling the fabric. Unfortunately, at the system level, a misconfigurations leaves software exposed to processor vulnerabilities and slowdowns due to false firmware activations. It is important to note that because MAGICCARPET operates below software, software attacks can-not leverage MAGICCARPET to control a compromised host.

The main limitation for recovery is the amount of functional redundancy in the processor. Since MAGICCARPET recodes instructions from the software stack in an attempt to execute around the processor imperfection, recovery depends on the existence of another sequence of instructions that yields an identical ISA-level state without activating the same or any other malicious circuit. This limitation did *not* manifest in any of our experiments, but it does exist.

Finally, as with any reference monitor [54], MAGICCARPET can not enforce properties that require knowledge of more than the current trace of execution such as properties about determinism and non-interference [18].

IX. ADDITIONAL RELATED WORK

MAGICCARPET has multiple components that extend into several different lines of research; such as processor verification, patching processor bugs, detecting malicious circuits, and recovering from processor faults. This section frames MAGICCARPET with respect to related work in those areas.

A. Hardware verification

The formal verification of hardware is a mature field. The functional verification of small, mostly stateless components of hardware implementations against a specification written in a C-like language has been well studied [17], [28], [35], [38], [37], [39]. The verification task becomes more difficult as the hardware becomes more complex. Burch and Dill demonstrated how to verify a pipelined processor implementation against an unpipelined specification using an automated technique akin to symbolic execution in software [15] and many others have since expanded the technique to apply to more complex processors [14], [48], [56]. More powerful, but less automated techniques based on theorem proving have been studied and applied to specialized hardware modules such as the floating point functions [31] as well as the processor pipeline [57], [43]. Alternatively, more automated model-checking based techniques have also been explored [45]. While many advances have been made, full functional verification of an implementation of a modern processor core is still out of reach for today's technology. By monitoring a set of invariants, MAGICCARPET guarantees that the current execution meets a set of security properties without requiring complete verification of the processor.

B. Processor bugs

Theo de Raadt [24] linked Intel errata to potential security vulnerabilities in the OpenBSD operating system. Subsequent attacks [40], [26], [10] used errata as a foothold. To understand the magnitude of exploitable processor bugs, Hicks *et al.* identified and classified a set of processor errata that eluded testing and functional verification and can debase software protections [34]. We use this classification in our evaluation of MAGICCARPET to exemplify escaped and exploitable processor vulnerabilities.

One approach to detecting bugs that escape verification is adding redundant but diverse computations, much like an always-on version of MAGICCARPET's recovery firmware. DIVA [9] is a processor bug patching mechanism using redundant implementations of an instruction set internal to 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. It requires complex interconnections between the checker core and the processor and the checker core must be extremely simple to allow formal verification. These requirements preclude performance optimizations, thereby limiting system performance. In contrast, MAGICCARPET adds overhead only when the processor violates an assertion.

Another approach to handling processor bugs is signature-based detection of bug activations. Constantinides *et al.* [20] and Phoenix [53] use low-level hardware state (flip-flops) to form bug signatures and monitor the run time state of the processor for matches. Such approaches can detect more imperfections than MAGICCARPET, but: (1) they require two extra flip-flops for every flip-flop in the design, and (2) heavy contention on opcode flip-flops limit the number of bugs that can be monitored concurrently [32].

Semantic Guardians [60] is a third approach. These are assertions on low-level state that fire at runtime whenever the processor enters a state that was not functionally verified. This avoids the heavy contention for shared resources in signature-based approaches, as unverified states have independent assertion circuits. The challenge comes from the many states not seen during verification; building an assertion per unseen state is infeasible.

C. Design-time malicious circuits

Malicious designers [59] may add intentional bugs to the processor. King *et al.* [36] and Hicks *et al.* [33] demonstrate three possible system-level attacks, showing that it is possible for malicious circuits to be footholds for software-level attacks.

Unused Circuit Identification (UCI) [33] attempts to identify and remove suspicious circuits from processors by re-purposing test cases used in functional verification to find circuits that do not seem to affect the processor's behavior. FANCI [63] extends UCI's circuit identification idea by avoiding a binary notion of trust, which can be defeated [58], as well as removing the need to trust the verification test cases. FANCI explores the input space, counting the number times a circuit affects the state of the processor, generating a continuous notion of trust in a circuit, and exposing a tradespace where designers can trade more time at the design stage for increased trust in the design. General-purpose algorithms for bypassing UCI and FANCI analyses have since been developed [58], [66]. These defenses and attacks hint that the general problem of assuring that hardware is free from attack is equivalent to proving that the processor is free from bugs. MAGICCARPET is orthogonal to design-time testing approaches: MAGICCARPET allows processor designers to respond to processor vulnerabilities post-deployment, thus MAGICCARPET is able to respond to attacks missed by UCI and FANCI.

TrustNet and DataWatch[61] monitor communication between processor pipeline stages ensuring that, given the input, the output contains the correct amount of data. Although these mechanisms prevent significant classes of attacks, they succumb to some hardware-based attacks in the literature (*e.g.*,, the *memory redirection* attack from Hicks *et al.* [33]) that are carried out via data modifications rather than additions or subtractions of data. As shown in Section VII, MAGICCARPET can check for too much, too little, or incorrect data.

Recently, researchers have proposed protecting software from malicious hardware by using assertions to focus on attack effect rather than properties of the attack implementation. Security Checkers [11], [12] and SPECS [34] add assertions to the processor that monitor for security violations at run time. MAGICCARPET extends this notion of monitoring security properties at run time to an adversarial model in which an attacker is aware of the defenses and attempts to bypass them. With Security Checkers and SPECS any escaped attack compromises the system for the life of the system. Contrast this with MAGICCARPET, whose entire goal is to respond to the escaped processor vulnerabilities at run time. MAGICCARPET also diverges from Security Checkers and SPECS with its ability to allow unaltered software to continue execution on vulnerable hardware and through the pervasive use of formal methods.

D. Software-level assertions

Software-level assertions have been proposed to detect processor defects. SWAT [41], [52] is a system that periodically checks for anomalous software behavior, *e.g.,* deadlock or kernel panic, as a sign of processor errors. While over 70% of randomly inserted single-bit faults are covered, SWAT is unable to detect any of the errata-based attacks of Section VII.

Pham *et al.* [49], [50] use the hardware invariants available, *e.g.,* in AMD-V's [7] *Hardware-Assisted Virtualization* (HAV), to implement HyperTap, a hypervisor scheme to provide reliability and security monitoring. HyperTap's invariants are used to enforce properties in the software stack above the hardware, which is trusted. It is clear that software requires assistance from hardware to protect against a wide range of security vulnerabilities, including processor bugs [8].

E. Recovery

Recovery using checkpoint/rollback is well-explored and low overhead solutions exist [55], [22], [27], but re-execution is inadequate to move software state past a malicious circuit.

Software simulation of portions of an ISA is used in microprocessors lacking hardware floating point units [2]. Narayanasamy *et al.* [44] extend this idea to processor bug patching by using simple, *ad hoc*, instruction rewriting routines to avoid triggering the bug. They sketch software-based recovery strategies for five errata. BlueChip [33] is a recovery mechanism targeted at malicious circuits implemented as a Linux device driver. It further extends the Narayanasamy *et al.* approach. While this prior work uses software to recover from processor defects, three differences exist. First, we formally verify MAGICCARPET's recovery routines. Second MAGICCARPET implements recovery as a firmware layer, preserving the ISA abstractions expected by software and allowing full-system recovery, in contrast to BlueChip. Third, MAGICCARPET's hardware support for recursive activations and wider range of recovery routines greatly improve the scope of recovery.

X. CONCLUSION

MAGICCARPET is a new defense against hardware-based security exploits. MAGICCARPET adds *verified* hardware to a processor to detect run-time failures of security invariants. Detections trigger a transfer of control to *verified* recovery firmware that briefly simulates software using equivalent instruction sequences, enabling safe forward progress on the vulnerable hardware. MAGICCARPET's configuration fabric allows processor designers to respond to newly discovered vulnerabilities at run time. Experiments show that MAGICCARPET is effective at detecting and recovering from a range of real-world processor vulnerabilities.

MAGICCARPET effectively protects software from exploitable hardware by pushing the task of dealing with the vulnerability to a higher, more flexible, and more verifiable level. We made pervasive use of formal methods when implementing MAGICCARPET hardware and firmware. We argue that this style of design, where we focus our formal verification efforts on the hardware and software responsible for monitoring the security relevant portion of the processor, represents a viable alternative to full functional correctness. The bottom line is that we can make stronger security guarantees for a processor *with a significantly smaller verification effort than would be required for full functional correctness.*

References

[1] Anonymous citation for blind review.

[2] ARMv4 Instruction Set, Issue C, April 1998.

[3] MIPS Technologies. MIPS R4000PC/SC Errata, Processor Rev. 2.2 and 3.0, May 1994.

[4] Revision Guide for AMD Athlon 64 and AMD Opteron Processors. Technical report, Advanced Micro Devices, August 2005.

[5] Intel Core 2 Extreme Processor X6800 and Intel Core 2 Duo Desktop Processor E6000 and E4000 Sequence – Specification Update. Technical report, Intel Corporation, May 2008.

[6] Accellera Systems Initiative. Open verification library working group. http://www.accellera.org/activities/committees/ovl.

[7] AMD. AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual, May 2005.

[8] D. Arora, A. Raghunathan, S. Ravi, and N. K. Jha. Enhancing security through hardware-assisted run-time validation of program data properties. In *Proc. CODES+ISSS*, pages 190–195. ACM, 2005.

[9] T. M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proc. ACM/IEEE MICRO*, pages 196–207, Haifa, Israel, November 1999.

[10] E. Biham, Y. Carmeli, and A. Shamir. Bug attacks. In *Proc. CRYPTO*, pages 221–240, 2008.

[11] M. Bilzor, C. Irvine, T. Huffmire, and T. Levin. Security Checkers: Detecting Processor Malicious Inclusions at Runtime. In *Proc. IEEE HOST*, pages 34–39, 2011.

[12] M. Bilzor, C. Irvine, T. Huffmire, and T. Levin. Evaluating security requirements in a general-purpose processor by combining assertion checkers with code coverage. In *Proc. IEEE HOST*, pages 49–54, 2012.

[13] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, Sept. 1992.

[14] J. R. Burch. Techniques for verifying superscalar microprocessors. In *Proceedings of the 33rd annual Design Automation Conference*, pages 552–557. ACM, 1996.

[15] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification*, pages 68–80. Springer, 1994.

[16] J. Chang, G. A. Reis, and D. I. August. Automatic instruction-level software-only recovery. In *Proc. DSN*, pages 83–92, 2006.

[17] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 308–311. IEEE Computer Society Press, 2003.

[18] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[19] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs 2009*, pages 23–42, 2009.

[20] K. Constantinides, O. Mutlu, and T. Austin. Online Design Bug Detection: RTL Analysis, Flexible Mechanisms, and Evaluation. In *Proc. MICRO*, pages 282–293, 2008.

[21] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal Verification of Information Flow Security for a Simple ARM-Based Separation Kernel. In *Proc. CCS*, 2013.

[22] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *Proc. ISCA*, pages 497–508, 2010.

[23] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS 2008*, pages 337–340, 2008.

[24] T. de Raadt. Intel core 2. openbsd-misc mailing list, June 2007.

[25] Digilent Inc. Xupv5 development board. http://www.digilentinc.com/Products/Detail.cfm?NavTop=2&NavSub=599&Prod=XUPV5.

[26] L. Duflot. CPU Bugs, CPU Backdoors and Consequences on Security. In *Proc. ESORICS*, pages 580–599, 2008.

[27] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August. Encore: Low-cost, Fine-grained Transient Fault Recovery. In *Proc. MICRO*, pages 398–409, 2011.

[28] X. Feng and A. J. Hu. Early cutpoint insertion for high-level software vs. RTL formal combinational equivalence verification. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1063–1068. ACM, 2006.

[29] H. Foster, K. Larsen, and M. Turpin. Introduction to the New Accellera Open Verification Library. In *DVCon06: Proceedings of the Design and Verification Conference and exhibition*, 2006.

[30] I. Hadzic, S. K. Udani, and J. M. Smith. FPGA Viruses. In *Proc. FPL*, pages 291–300. Springer, 1999.

[31] J. Harrison. Formal verification of floating point trigonometric functions. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 217–233, London, UK, UK, 2000. Springer-Verlag.

[32] M. Hicks. *Practical Systems for Overcoming Processor Imperfections*. PhD thesis, University of Illinois Urbana-Champaign, April 2013.

[33] 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 *Proc. IEEE Security and Privacy*, pages 159–172, 2010.

[34] 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 *Proc. ASPLOS*, 2015. to appear.

[35] A. J. Hu. High-level vs. RTL combinational equivalence: An introduction. In *International Conference on Computer Design (ICCD)*, pages 274–279. IEEE, 2007.

[36] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and implementing malicious hardware. In *Proc. USENIX LEET*, April 2008.

[37] A. Koelbl, J. R. Burch, and C. Pixley. Memory modeling in ESL-RTL equivalence checking. In *ACM/IEEE Design Automation Conference (DAC)*, pages 205–209. IEEE, 2007.

[38] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley. Solver technology for system-level to RTL equivalence checking. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 196–201, 2009.

[39] A. Koelbl and C. Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming*, 33(6):645–666, 2005.

[40] S. Lemon. Researcher to Demonstrate Attack Code for Intel Chips. http://www.pcworld.com/article/148353/security.html, July 2008.

[41] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *Proc. ASPLOS*, pages 265–276. ACM, 2008.

[42] K. McMillan. The Cadence SMV Model Checker. http://www.kenmcmil.com/smv.html, 2007.

[43] S. P. Miller and M. Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *Proceedings of the 1st Workshop on Industrial-Strength Formal Specification Techniques*, WIFT '95, pages 2–, Washington, DC, USA, 1995. IEEE Computer Society.

[44] S. Narayanasamy, B. Carneal, and B. Calder. Patching processor design errors. In *Proc. IEEE ICCD*, pages 491–498, October 2006.

[45] E. Nurvitadhi, J. Hoe, T. Kam, and S. Lu. Integrating formal verification and high-level processor pipeline synthesis. In *Proceedings of the 9th Symposium on Application Specific Processors (SASP)*. IEEE, 2011.

[46] OpenCores.org. OpenRISC OR1200 processor. http://opencores.org/or1k/OR1200_OpenRISC_Processor.

[47] OpenRISC.net. OpenRISC.net mailing list. http://lists.openrisc.net.

[48] V. Patankar, A. Jain, and R. Bryant. Formal verification of an ARM processor. In *Proceedings of the Twelfth International Conference on VLSI Design*, pages 282–287, 1999.

[49] C. Pham, Z. Estrada, P. Cao, Z. Kalbarczyk, and R. Iyer. Reliability and Security Monitoring of Virtual Machines Using Hardware Architectural Invariants. In *Proc. DSN*, 2014.

[50] C. Pham, Z. J. Estrada, P. Cao, Z. Kalbarczyk, and R. K. Iyer. Building Reliable and Secure Virtual Machines Using Architectural Invariants. *IEEE Security and Privacy*, 12(5):82–85, Sept 2014.

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

[52] S. Sahoo, M.-L. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *Proc. DSN*, pages 70–79, 2008.

[53] S. R. Sarangi, A. Tiwari, and J. Torrellas. Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware. In *Proc. MICRO*, pages 26–37, 2006.

[54] F. B. Schneider. Enforceable Security Policies. *ACM TISSEC*, 3(1):30–50, Feb. 2000.

[55] 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 *Proc. ISCA*, pages 123–134, Anchorage, Alaska, May 2002.

[56] S. Srinivasan and M. Velev. Formal verification of an Intel XScale processor model with scoreboarding, specialized execution pipelines, and impress data-memory exceptions. In *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*. ACM and IEEE, 2003.

[57] M. Srivas and S. Miller. Applying formal verification to a commercial microprocessor. In *Proceedings of the Asia and South Pacific Design Automation Conference*, 1995.

[58] C. Sturton, M. Hicks, D. Wagner, and S. T. King. Defeating UCI: Building stealthy and malicious hardware. In *Proc. IEEE Security and Privacy*, pages 64–77, 2011.

[59] Trust-Hub. https://www.trust-hub.org/.

[60] I. Wagner and V. Bertacco. Engineering Trust with Semantic Guardians. In *Proc. DATE*, pages 743–748, 2007.

[61] A. Waksman and S. Sethumadhavan. Tamper Evident Microprocessors. In *Proc. IEEE Security and Privacy*, 2010.

[62] A. Waksman and S. Sethumadhavan. Silencing hardware backdoors. In *Proc. IEEE Security and Privacy*, 2011.

[63] A. Waksman, M. Suozzo, and S. Sethumadhavan. FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis. In *Proc. CCS*, pages 697–708, 2013.

[64] Xen.org security team. [xen-announce] xen security advisory 7 (cve-2012-0217) - pv. Xen mailing list, June 2012.

[65] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proc. IEEE Security and Privacy*, pages 79 –93, May 2009.

[66] J. Zhang, F. Yuan, and Q. Xu. DeTrust: Defeating hardware trust verication with stealthy implicitly-triggered hardware trojans. In *Conference on Computer and Communications Security*. ACM, 2014.

XI. Appendix A: Invariants

The tables here are from [34] and re-printed with the authors' permission.

ID	**MagicCarpet Invariant**	**OVL Assertion-based 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	Return from exception 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 III: Invariants developed to protect against security-critical processor bugs

ID	**MagicCarpet 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 IV: Detailed descriptions of invariants; IDs correspond to the invariant IDs in Table III.