5-2010

Overcoming and Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically

Matthew Hicks
University of Illinois at Urbana-Champaign

Murph Finnicum
University of Illinois at Urbana-Champaign

Samuel T. King
University of Illinois at Urbana-Champaign

Milo Martin
University of Pennsylvania, milom@cis.upenn.edu

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

Recommended Citation

Matthew Hicks, Murph Finnicum, Samuel T. King, Milo Martin, and Jonathan M. Smith, "Overcoming and Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically", . May 2010.

Hicks, M., Finnicum, M., King, S., Martin, M., & Smith, J., Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically, *31st IEEE Symposium on Security and Privacy*, May 2010, doi: 10.1109/SP.2010.18

Overcoming and Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically

Abstract

The computer systems security arms race between attackers and defenders has largely taken place in the domain of software systems, but as hardware complexity and design processes have evolved, novel and potent hardware-based security threats are now possible. This paper presents a hybrid hardware/software approach to defending against malicious hardware. We propose BlueChip, a defensive strategy that has both a design-time component and a runtime component. During the design verification phase, BlueChip invokes a new technique, unused circuit identification (UCI), to identify suspicious circuitry—those circuits not used or otherwise activated by any of the design verification tests. BlueChip removes the suspicious circuitry and replaces it with exception generation hardware. The exception handler software is responsible for providing forward progress by emulating the effect of the exception-generating instruction in software, effectively providing a detour around suspicious hardware. In our experiments, BlueChip is able to prevent all hardware attacks we evaluate while incurring a small runtime overhead.

Disciplines

Computer Sciences

Comments

Hicks, M., Finnicum, M., King, S., Martin, M., & Smith, J., Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically, *31st IEEE Symposium on Security and Privacy*, May 2010, doi: 10.1109/SP.2010.18

Appears in *Proceedings of the 31st IEEE Symposium on Security & Privacy (Oakland)*, May 2010

Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically

Matthew Hicks, Murph Finnicum, Samuel T. King
University of Illinois at Urbana-Champaign

Milo M. K. Martin, Jonathan M. Smith
University of Pennsylvania

Abstract

The computer systems security arms race between attackers and defenders has largely taken place in the domain of software systems, but as hardware complexity and design processes have evolved, novel and potent hardware-based security threats are now possible. This paper presents a hybrid hardware/software approach to defending against malicious hardware.

We propose BlueChip, a defensive strategy that has both a design-time component and a runtime component. During the design verification phase, BlueChip invokes a new technique, *unused circuit identification (UCI)*, to identify suspicious circuitry—those circuits not used or otherwise activated by any of the design verification tests. BlueChip removes the suspicious circuitry and replaces it with exception generation hardware. The exception handler software is responsible for providing forward progress by emulating the effect of the exception-generating instruction in software, effectively providing a detour around suspicious hardware. In our experiments, BlueChip is able to prevent all hardware attacks we evaluate while incurring a small runtime overhead.

1 Introduction

Modern hardware design processes closely resemble the software design process. Hardware designs consist of millions of lines of code and often leverage libraries, toolkits, and components from multiple vendors. These designs are then "compiled" (synthesized) for fabrication. As with software, the growing complexity of hardware designs creates opportunities for hardware to become a vehicle for malice. Recent work has demonstrated that small malicious modifications to a hardware-level design can compromise the security of the entire computing system [22].

Malicious hardware has two key properties that make it even more damaging than malicious software. First, hardware presents a more persistent attack vector. Whereas software vulnerabilities can be fixed via software update patches or reimaging, fixing well-crafted hardware-level vulnerabilities would likely require physically replacing the compromised hardware components. A hardware recall similar to Intel's Pentium FDIV bug (which cost 500 million dollars to recall five million chips) has been estimated to cost many billions of dollars today [7]. Furthermore, the skill required to replace hardware and the rise of deeply embedded systems ensure that vulnerable systems will remain in active use after the discovery of the vulnerability. Second, hardware is the lowest layer in the computer system, providing malicious hardware with control over the software running above. This low-level control enables sophisticated and stealthy attacks aimed at evading software-based defenses.

Such an attack might use a special, or unlikely, event to trigger deeply buried malicious logic which was inserted during design time. For example, attackers might introduce a sequence of bytes into the hardware that activates the malicious logic. This logic might escalate privileges, turn off access control checks, or execute arbitrary instructions, providing a path for the malefactor to take control of the machine. The malicious hardware thus provides a *foothold* for subsequent system-level attacks.

In this paper we present the design, implementation, and evaluation of BlueChip, a hybrid design-time/runtime system for detecting and neutralizing malicious circuits. During the design phase, BlueChip flags as suspicious, any unused circuitry (any circuit not activated by any of the many design verification tests) and deactivates them. However, these seemingly suspicious circuits might actually be part of a legitimate circuit within the design, so BlueChip inserts circuitry to raise an exception whenever one of these suspicious circuits would have been activated. The exception handler software is responsible for emulating hardware instructions to allow the system to continue execution. BlueChip's overall goal is to push the complexity of coping with malicious hardware up to a higher, more flexible, and adaptable layer in the system stack.

The contributions of this paper are:

- We present the BlueChip system (Sections 3 and 4), which automatically removes potentially malicious circuits from a hardware design and uses low-level software to emulate around removed hardware.

- We propose an algorithm (Section 5), called *unused circuit identification*, for automatically identifying circuits that avoid affecting outputs during design verification. We demonstrate its feasibility (Section 6) for use in addressing the problem of detecting malicious hardware.

- We demonstrate (Sections 7, 8, and 9), using fully-tested malicious hardware modifications as test cases on a SPARC processor implementation operating on an FPGA, that: (1) the system successfully prevents three different malicious hardware modifications, and (2) the performance effects (and hence the overhead) of the system are small.

2 Motivation and attack model

This paper focuses on the problem of malicious circuits introduced during the hardware design process. Today's complicated hardware designs are increasingly vulnerable to the undetected insertion of malicious circuitry to create a hardware trojan horse. In other domains, examples of this general type of intentional insertion of malicious functionality include compromises of software development tools [26], system designers inserting malicious source code intentionally [8, 20, 21], compromised servers that host modified source code [11, 12], and products that come pre-installed with malware [1, 4, 25]. Such attacks introduce little risk of punishment, because the complexity of modern systems and prevalence of unintentional bugs makes it difficult to prove malice or to correctly attribute the problem to its source [27].

More specifically, our threat model is that a rogue designer covertly adds trojan circuits to a hardware design. We focus on two possible scenarios for such rogue insertion. First, one or more disgruntled employees at a hardware design company surreptitiously and intentionally inserts malicious circuits into a design prior to final design validation with the hope that the changes will evade detection. The malicious hardware demonstrated by King *et al.* [22] support the plausibility of this scenario, in that only small and localized changes (*e.g.*, to a single hardware source file) are sufficient for creating powerful malicious circuits designed for bootstrapping larger system-level attacks. We call such malicious circuits *footholds*, and such footholds persist even after malicious software has been discovered and removed, giving attackers a permanent vector into a compromised system.

The second scenario is enabled by the trend toward "softcores" and other pre-designed hardware IP (intellectual property) blocks. Many system-on-chip (SoC) designs aggregate subcomponents from existing commercial or open-source IP. Although generally trusted, these third-party IP blocks may not be trustworthy. In this scenario, an attacker can create new IP or modify existing IP blocks to add malicious circuits. The attacker then distributes or licenses the IP in the hope that some SoC creator will incorporate it and include it in a fabricated chip. Although the SoC creator will likely perform significant design verification focused on finding design bugs, traditional black-box design verification is unlikely to reveal malicious hardware.

In either scenario, the attacker's motivation could be financial or general malice. If the design modification remains undetected by final design validation and verification, the malicious circuitry will be present in the manufactured hardware that is shipped to customers and integrated into computing systems. The attacker has achieved this without the resources necessary to actually fabricate a chip or otherwise attacking the manufacturing and distribution supply chain. We assume that only one or a few individuals are acting maliciously (*i.e.*, not the entire design team) and that these individuals are unable to compromise the final end-to-end design verification and validation process, which is typically performed by a distinct group of engineers.

Our approach to detecting insertions of malicious hardware assumes analysis at the level of a hardware netlist or hardware description language (HDL) source. In the two scenarios outlined, this assumption is reasonable, as (1) design validation and verification is primarily performed at this level and (2) softcore IP blocks are often distributed in HDL or netlist form.

We assume the system software is trustworthy and non-malicious (although the malicious hardware may attempt to subvert the overlying software layers).

3 The BlueChip approach

Our overall BlueChip architecture is shown in Figure 1. In the first phase of operation, BlueChip analyzes the circuit's behavior during design verification to identify candidate circuits that might be malicious. Once BlueChip identifies a suspect circuit, BlueChip automatically removes the circuit from the design. Because BlueChip might remove legitimate circuits as part of the transformation, it inserts logic to detect if the removed circuits would have been activated, and triggers an exception if the hardware encounters this condition during runtime. The hardware delivers this exception to the BlueChip software layer. The exception handling software is responsible for recovering from the fault and advancing the computation by emulating the instruction that was exe-

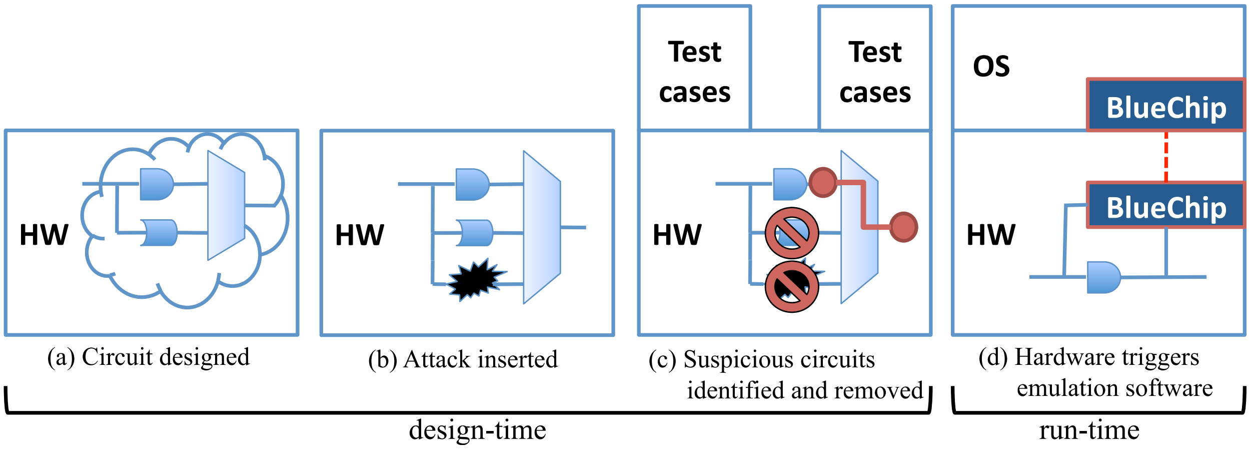


Figure 1: Overall BlueChip architecture. This figure shows the overall flow for BlueChip where (a) designers develop hardware designs and (b) a rogue designer inserts malicious logic into the design. During design verification phase, (c) BlueChip identifies and removes suspicious circuits and inserts runtime hardware checks. (d) During runtime, these hardware checks invoke software exceptions to provide the BlueChip software an opportunity to advance the computation by emulating instructions, even though BlueChip may have removed legitimate circuits.

cuting when the exception occurred. BlueChip pushes much of the complexity up to the software layer, allowing defenders to rapidly refine defenses, turning the permanence of the hardware attack into a disadvantage for attackers.

BlueChip can operate in spite of removed hardware because the removed circuits operate at a lower layer of abstraction than the software emulation layer responsible for recovery. BlueChip software does not emulate the removed hardware directly. Instead, BlueChip software emulates the effects of removed hardware using a simple, high-level, and implementation-independent specification of hardware, *i.e.*, the processor's instruction-set-architecture specification. The BlueChip software emulates the effects of the removed hardware by emulating one or more instructions, updating the processor registers and memory values, and resuming execution. The computation can generally make forward progress despite the removed hardware logic, although software emulation of instructions is slower than normal hardware execution.

In some respects our overall BlueChip system resembles floating point instruction emulation for processors that omit floating point hardware. If a processor design omits floating point unit (FPU) hardware, floating point instructions raise an exception that the OS handles. The OS can emulate the effects of the missing hardware using available integer instructions. Like FPU emulation, BlueChip uses software to emulate the effects of missing hardware using the available hardware resources. However, the hardware BlueChip removes is *not* necessarily associated with specific instructions and can trigger BlueChip exceptions at unpredictable states and events, presenting a number of unique challenges that we address in Section 4.

4 BlueChip design

This section describes the design of BlueChip. We discuss the BlueChip hardware component (Section 4.1), the BlueChip software component (Section 4.2), and possible alternative architectures (Section 4.3). Section 5 discusses our algorithm for identifying suspicious circuits and Section 6 describes how BlueChip uses these detection results to modify the hardware design.

We present general requirements for applying BlueChip to hardware and to software, but we describe our specific design for a modified processor and recovery software running within an operating system.

4.1 BlueChip hardware

To apply BlueChip techniques, a hardware component must be able to meet three general requirements. First, BlueChip requires a hardware exception mechanism for passing control to the software. Second, BlueChip must prevent modified hardware state from committing when the hardware triggers a BlueChip exception. Third, to enable recovery the hardware must provide software access to hardware state, such as processor register values and other architecturally visible state.

Processors are well suited to meet the requirements for BlueChip because they already have many of the mechanisms BlueChip requires. Processors provide easy access to architecturally visible states to enable context switching, and processors have existing exception delivery mechanisms that provide a convenient way to pass control to a software exception handler. Also, most processors support precise exceptions that include a lightweight recovery mechanism to prevent committing any state associated with the exception. As a result, we use existing exception handling mechanisms within the processor to deliver BlueChip exceptions.

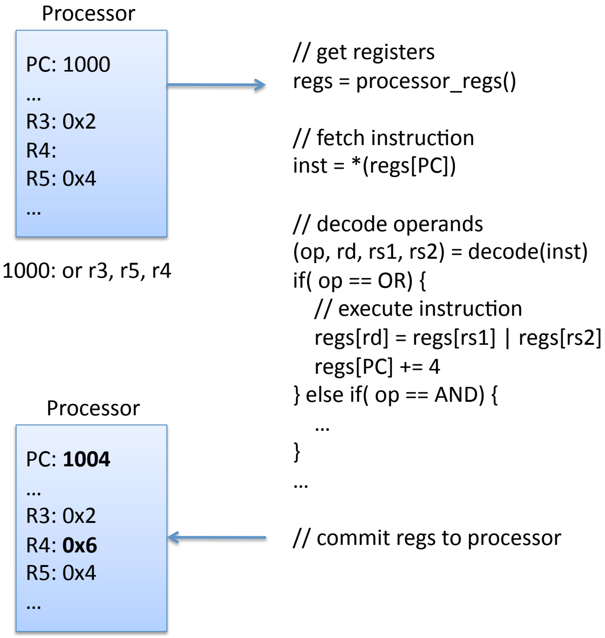
One modification we make to current exception semantics is to assert BlueChip exceptions immediately instead of associating them with individual instructions. In current processors, many exceptions are associated with a specific instruction that caused the fault. However, in BlueChip it is often unclear which individual instruction would have triggered the removed logic. Thus, BlueChip asserts the exception immediately, flushes the pipeline, and passes control to the BlueChip software.

4.2 BlueChip software

The BlueChip software is responsible for recovering from BlueChip hardware exceptions and providing a mechanism for system forward progress. This responsibility presents unusual design challenges for the BlueChip software because it runs on a processor that has had some portions of the design removed, therefore some features of the processor may be unavailable to the BlueChip exception handler software.

To handle BlueChip exceptions, BlueChip uses a recovery technique where the BlueChip software emulates faulting instructions to carry out the computation. The basic emulation strategy is similar to an instruction-by-instruction emulator, where for each instruction, BlueChip software reads the instruction from memory, decodes the instruction, calculates the effects of the instruction, and commits the register and memory changes (Figure 2). By emulating instructions in software, BlueChip skips past the instructions that use the removed hardware, duplicating their effects in software, providing the opportunity for the system to continue making forward progress despite the missing circuits.

One problem with our first implementation of this basic strategy was that our emulation routines sometimes depended on removed hardware, thus causing unrecoverable recursive BlueChip exceptions. For example, the "shadow-mode attack" (Section 7) uses a "bootstrap trigger" that initiates the attack. The bootstrap trigger circuit monitors the values going into the data cache and enables the attack once it observes a specific value being stored in memory. This specific value will always trigger the attack regardless of the previous states and events in the system. After BlueChip identified and removed the attack circuits, the BlueChip hardware triggered an exception whenever the software attempted to store the attack value to a memory location. Our first implementation of the `store` emulation code simply re-issued a `store` instruction with the same address and value to emulate the effects of the removed logic, thus creating an unrecoverable recursive exception.



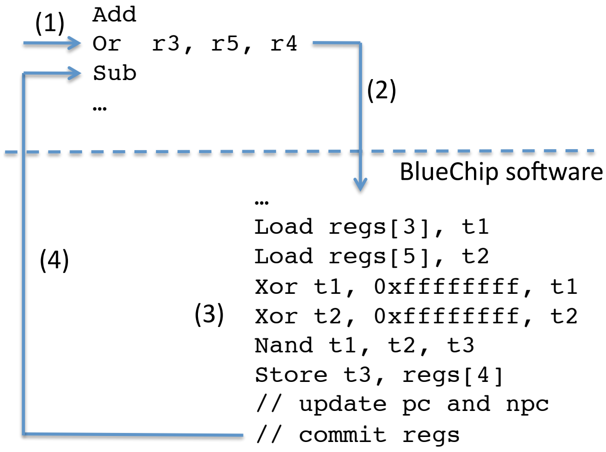
```
// get registers
regs = processor_regs()

// fetch instruction
inst = *(regs[PC])

// decode operands
(op, rd, rs1, rs2) = decode(inst)
if( op == OR) {
   // execute instruction
   regs[rd] = regs[rs1] | regs[rs2]
   regs[PC] += 4
} else if( op == AND) {
   …
}
…

// commit regs to processor
```

Figure 2: Basic flow for an instruction-by-instruction emulator. This figure shows how a software emulator can calculate the changes to processor registers induced by an `or` instruction.

To avoid unrecoverable recursive BlueChip exceptions, we emulate around faulting instructions by producing semantically equivalent results while avoiding BlueChip exception states. For ALU operations, we map the emulated instructions to an alternative set of ALU operations and equivalent, but different, operand values. For example, we implement `or` emulation using a series of `xor`, and `nand` instructions rather than executing an `or` to perform OR operations (Figure 3). For `load` and `store` instructions we have somewhat less flexibility because these instructions are the sole means for performing I/O operations that access off-processor memory. However, we do perform some transformations, such as emulating word sized accesses using byte sized accesses and vice versa.

4.3 Alternative designs

In this section we discuss other possible designs and some of the trade offs inherent in their design decisions.



```
Add
Or  r3, r5, r4
Sub
…
```

```
…
Load regs[3], t1
Load regs[5], t2
Xor t1, 0xffffffff, t1
Xor t2, 0xffffffff, t2
Nand t1, t2, t3
Store t3, regs[4]
// update pc and npc
// commit regs
```

Figure 3: BlueChip emulation. This figure shows how BlueChip emulates around removed hardware. First, (1) the software executes an `or` instruction, (2) which causes a BlueChip exception. This exception is handled by the BlueChip software, (3) which emulates the `or` instruction using `xor` and `nand` instructions (4) before returning control to the next instruction in the program.

BlueChip delivers exceptions using existing processor exception handling mechanisms. One alternative could be adding new hardware to deliver exceptions to the BlueChip software component. In our current design, BlueChip is constrained by the semantics of the existing exception handling mechanisms and cannot deliver exceptions when software disables interrupts.

An alternative approach could have been to add extra registers and logic to the processor to allow BlueChip to save state and recover from BlueChip exceptions, even when the software disables interrupts. However, this additional state and logic would have required encoding several implementation-specific details of the hardware design into BlueChip, potentially making it more difficult to insert BlueChip logic automatically. Given the infrequency of disabling interrupts for long periods in modern commodity operating systems, we decided to use existing processor exception delivery mechanisms. If a particular choice of hardware and software makes this unacceptable, there are several straightforward approaches to addressing this issue, such as using a hypervisor with some additional support for BlueChip.

BlueChip emulates around BlueChip exceptions by using different instructions to emulate computations that depend on hardware removed by BlueChip. In our current design we implement this emulation technique manually for all instructions in the processor's instruction set. However, we still rely on portions of the OS and exception handling code to save and restore the system states we emulate around. It might be possible for these instructions to inadvertently invoke an unrecoverable BlueChip exception by executing an instruction that causes a BlueChip exception. One way to avoid unrecoverable BlueChip exceptions could be to modify the compiler to emit only a small set of Turing complete instructions for the BlueChip software, such as `nand`, `load`, and `store` instructions. Then we could focus our testing or formal methods efforts on this subset of the instruction set to decrease the probability of an unrecoverable BlueChip exception. This technique would likely make the BlueChip software slower because it potentially uses more instructions to carry out equivalent computations, but it could decrease the occurrence of unrecoverable BlueChip exceptions.

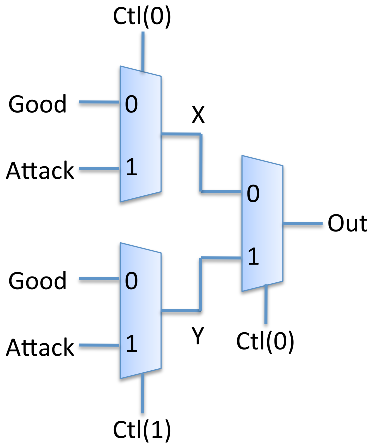
5 Detecting suspicious circuits

This section describes our detection algorithm for identifying suspicious circuits automatically within a hardware design. We focus on automatically detecting potentially malicious logic embedded within the HDL source code of a design, and we perform our detection during the design phase of the hardware design life cycle.

Our goal is to develop an algorithm that identifies malicious circuits without identifying benign circuits. In addition, our technique should be difficult for an attacker to avoid, and it should identify potentially malicious code automatically without requiring the defender to develop a new set of design verification tests specifically for our new detection algorithm.

Hardware designs often include extensive design verification tests that designers use to verify the functionality of a component. In general, test cases use a set of inputs and verify that the hardware circuit outputs the expected results. For example, test cases for processors use a sequence of instructions as the input, with the processor registers and system memory as outputs.

Our approach is to use design verification tests to help detect attack circuits. If an attack circuit contaminates the output for a test case, the designer would know that the circuit is operating out-of-spec, potentially detecting the attack. However, recent research has shown how hardware attacks can be implemented using small circuits that are designed not to trigger during routine testing [22]. This evasion technique works by guarding the attack circuit with triggering logic that enables the attack only when it observes a specific sequence of events or a specific data value (*e.g.*, the attack triggers only when the hardware encounters a predefined 128-bit value). This attack-hiding technique works because malicious hardware designers can avoid perturbing outputs during testing by hiding deep within the vast state



```
X <= (Ctl(0) = '0') ? Good : Attack
Y <= (Ctl(1) = '0') ? Good : Attack
Out <= (Ctl(0) = '0') ? X : Y
```

Figure 4: Circuit diagram and HDL source code for a mux that can pass code coverage testing without enabling the attack. This figure shows how a well-crafted mux can pass coverage tests when the appropriate control states (Ctl(0) and Ctl(1)) are triggered during testing. Control states 00, 01, and 10 will fully cover the circuit without triggering the attack condition.

```
// step one: generate data-flow graph
// and find connected pairs
pairs = {connected data-flow pairs}

// step two: simulate and try to find
// any logic that does not affect the
// data-flow pairs
foreach simulation clock cycle
  foreach pair in pairs
    if the sink and source not equal
      remove the pair from the pairs set
```

Figure 5: Identifying potentially malicious circuits using our UCI algorithm.

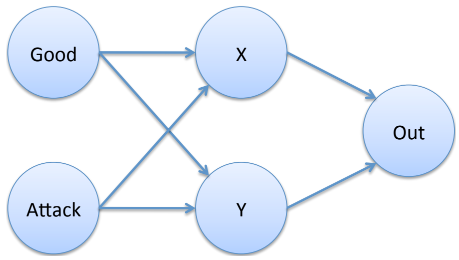


Figure 6: Data-flow graph for mux replacement circuit.

space of a design,[1] but can still enable attacks in the field by inducing the trigger sequence. Our proposal is to consider circuits suspicious whenever they are included in a design but do *not* affect any outputs during testing.

5.1 Straw-man approach: code coverage

One possible approach to identifying potentially malicious circuits could be to use code coverage. Code coverage is defined as the percentage of lines of code that are executed, out of those possible. Because attackers will likely try to avoid affecting outputs during testing, highlighting uncovered lines of code seems like a viable approach to identifying potentially malicious circuits.

An attacker can easily craft circuits that are covered completely by testing, but never trigger an attack. For example, Figure 4 shows a multiplexer (mux) circuit that can be covered fully without outputting the attack value. If the verification test suite includes control states 00, 01, and 10, all lines of code that make up the circuit will be covered, but the output will always be "Good". We apply this evasion technique for some of the attacks we evaluated (Section 7) and find that it does evade code coverage detection.

Although code coverage can complicate the attacker's task of avoiding testing, this technique can be defeated because code coverage misses the fundamental property of malicious circuits: attackers are likely to avoid affecting outputs during testing, otherwise they would be caught. Instead, what defenders need is a technique that zeros in on this property to identify potentially malicious circuits more reliably.

5.2 Unused circuit identification

This section describes our algorithm, called unused circuit identification (UCI), for identifying potentially malicious circuits at design time. Our technique focuses on identifying portions of the circuit that do not affect outputs during testing.

To identify potentially malicious circuits, our algorithm performs two steps (Figure 5). First, UCI creates a data-flow graph for our circuit (Figure 6). In this graph, nodes are signals (wires) and state elements; edges indicate data flow between the nodes. Based on this data-flow graph, UCI generates a list of all signal pairs, or *data-flow pairs*, where data flows from a source signal to a sink signal. This list of data-flow pairs includes both

[1] A processor with 16 32-bit registers, a 16k instruction cache, a 64k data cache, and 300 pins has *at least* 2^{655872} states, and up to 2^{300} transition edges.

direct dependencies (*e.g.*, (Good, X) in Figure 6) and indirect dependencies (*e.g.*, (Good, Out) in Figure 6).

Second, UCI simulates the HDL code using design verification tests to find the set of data-flow pairs where intermediate logic does not affect the data that flows between the source and sink signals. To test for this condition, at each simulation step UCI checks for inequality for each of our remaining data-flow pairs. If the elements of a pair are not equal, this implies, conservatively, that the logic in between the two pairs has an effect on the value, thus we remove pairs with unequal elements from our data-flow-pairs set. For registers, UCI accounts for latched data by maintaining a history of simulation values, allowing it make the appropriate comparison for tracking signal propagation.

After the simulation completes, UCI has a set of remaining data-flow pairs where the logic in between the pairs does not affect the signal value from source to sink. In other words, we could replace the intermediate logic with a wire, possible including some delay registers, and it would not affect the overall behavior of the circuit in any way for our design verification tests.

Consider how this algorithm works for the mux-replacement circuit shown in Figure 4:

1. UCI creates the initial set of data-flow pairs, which for our circuit is (Good,X), (Attack,X), (Good,Y), (Attack,Y), (Good,Out), (Attack,Out), (X,Out), and (Y,Out).

2. UCI considers the first simulation step where the control signals are 00 and the output is Good, X is Good, and Y is Good. This removes (Attack,X), (Attack,Y), and (Attack,Out).

3. UCI considers the second simulation step where the control signals are 01 and the output is Good, X is Good, and Y is Attack. This removes (Good,Y) and (Y,Out).

4. UCI considers the third simulation step where the control signals are 10 and the output is Good, X is Attack, and Y is Good. This removes (Good,X) and (X, Out).

5. UCI finishes the simulation and is left with (Good,Out) in the list of data-flow pairs where intermediate logic does not affect the signal propagation.

The resulting output from UCI for this example identifies the malicious circuit without identifying any additional signals. Because it systematically identifies circuits that avoid affecting outputs during testing, BlueChip connects the “Good” signal directly to the “Out” signal, thus removing the malicious elements from the design.

5.3 UCI limitations

In Section 9 we show that UCI successfully identifies the malicious circuits for the hardware-level footholds we developed. In this section we discuss ways an attacker could hide malicious circuits from UCI.

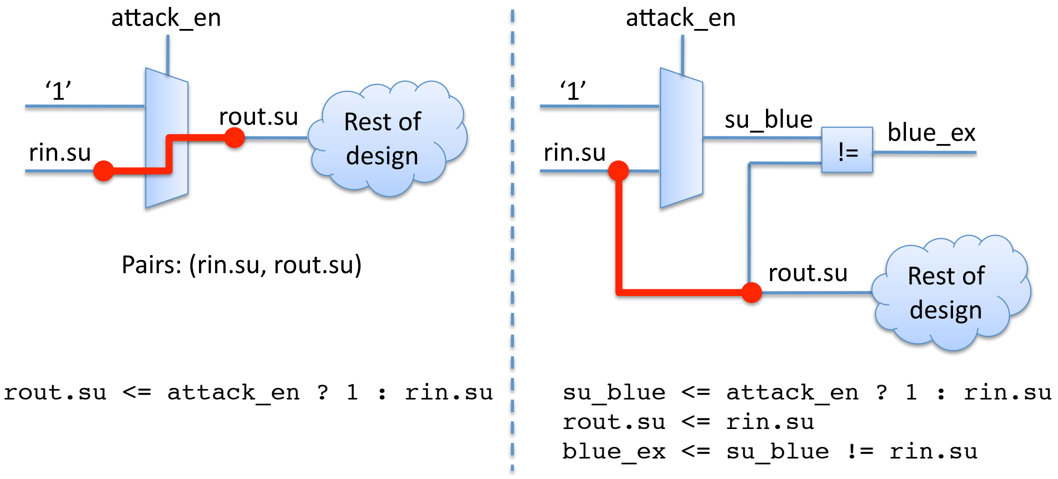
First, an attacker could include malicious test cases that check attack states incorrectly. By including these malicious test cases in the design verification test suite an attacker could fool the system designer into thinking that the out-of-spec modifications are in fact in-spec, thus slipping the attack past UCI. However, design verification tests work at a higher level of abstraction, making it easier for system designers to verify the results independently. In fact, the BlueChip software includes code for instruction-level emulation of the processor’s instruction set, and we use this emulation code on our test cases to verify that the test suite checks states correctly.

Second, an attacker could avoid UCI by crafting malicious circuits that affect unchecked outputs. Unchecked outputs could arise from incomplete test cases or from unspecified output states. For example, the memory model in the SPARC processor specification provides freedom for an implementation to relax consistency between processors in a multi-processor system [24]. This type of implementation-specific behavior could be used by an attacker who affects outputs that might be difficult for a testing program to check deterministically, thus causing malicious circuits to affect outputs and avoid our analysis. However, this evasion technique requires the attacker to trigger the attack during design verification, thus tests that check implementation-specific states and events could detect the foothold directly.

Third, to simplify the analysis of the HDL source, our current UCI implementation excludes mux control signals from the data-flow graph. If an attacker could use only mux control signals to modify architecturally visible states directly, UCI would miss the attack. However, this limitation is only an artifact of our current implementation and would likely be remedied if we included mux control signals in our analysis.

6 Using UCI results in BlueChip

This section discusses how BlueChip uses the results from UCI to eliminate the effects of suspicious circuits. UCI outputs a set of data-flow pairs, where each pair has a source element, a sink element, and a delay element. Conceptually, the source and the sink element can be connected directly by a wire containing delay number of registers, effectively short circuiting the signals with a delay line. This removes the effects of any intermediate logic between the source and the sink. BlueChip imple-



```
rout.su <= attack_en ? 1 : rin.su
```

```
su_blue <= attack_en ? 1 : rin.su
rout.su <= rin.su
blue_ex <= su_blue != rin.su
```

(a) UCI identifies attack

(b) BlueChip transforms HDL to remove attack

Figure 7: HDL code and circuit diagram for HDL transformations BlueChip makes to remove an attack from a design. This figure shows (a) the original design where an attacker can transition the processor into supervisor mode by asserting the `attack_en` signal. During design verification, UCI detects that the value for `rout.su` always equals `rin.su`, thus identifying the mux as a candidate for removal. Part (b) shows how BlueChip removes this logic by connecting `rin.su` to `rout.su` directly and adds exception notification logic to notify software of any inconsistencies at runtime.

ments this short-circuit by performing a source-to-source transformation on the design's HDL source code.

Once UCI generates a set of data-flow pairs, the pairs are fed into the BlueChip system (transformation pictured in Figure 7). BlueChip takes these pairs as an input and replaces the suspicious circuits by modifying the HDL code using three steps. First, BlueChip creates a backup version of each sink in the list of pairs. The backup version of a signal holds the value that would have been assigned to the signal in the original circuit. Second, BlueChip adds a new assignment to the original signal. The value assigned to the original signal is simply the value of the source element in the pair, creating a short circuit between the source and the sink. The rest of the design will see this short-circuited value, while the BlueChip exception generation hardware sees the backed-up version. The third and final step consists of adding the BlueChip exception generation logic to the source file. This circuit compares the backed-up value of the sink element with the source value. BlueChip generates an exception whenever any of these comparisons are not true. When a BlueChip exception occurs, it signals that the hardware was about to enter a state that was not seen during testing. From here, the BlueChip software is responsible for making forward progress.

The HDL transformation algorithm also must handle multiple data-flow pairs that share the same sink signal but have different sources. This situation could potentially cause problems for BlueChip because it is unclear which source signal to short-circuit to the original sink signal. Our solution is to pick one pair for the sink signal assignment, but include exception generation logic for all pairs that share the same sink element. This means that all violations are detected, but the sink may be shorted with the source that caused the exception. This untested state is safe because (1) BlueChip asserts exceptions immediately when detecting an inconsistency between the original design and the modified circuit, (2) BlueChip checks all data-flow pairs for inconsistencies, and (3) BlueChip leverages hardware recovery mechanisms to prevent the persistence of untrusted state modifications.

7 Malicious hardware footholds

This section describes the malicious hardware trojans we used to test the effectiveness of BlueChip. Prior work on developing hardware attacks focused on adding minimal additional logic gates as a starting point for a system-level attack [22]. We call this type of hardware mechanism a *foothold*. We explored three such footholds. The first foothold, called the *supervisor transition* foothold, enables an attacker to transition the processor into supervisor mode to escalate the privileges of user-mode code. The second foothold, called the *memory redirection* foothold, enables an attacker to read and write arbitrary virtual memory locations. The third foothold, called the *shadow mode* foothold, enables an attacker to pass control to invisible firmware located within the processor and take control of the system. Previous work has shown how these types of footholds can be used as part of a system-level attack to carry out high-level, high-value attacks, such as escalating privileges of a process or enabling attackers to login to a remote system automatically [22].

7.1 Supervisor transition foothold

Our supervisor transition foothold provides a hardware mechanism that allows unprivileged programs to transition the processor into supervisor mode. This transition grants access to privileged instructions and bypasses the usual hardware-enforced protections, allowing the attacker to escalate the privileges of an otherwise unprivileged process.

The malicious hardware is triggered when it observes a sequence of instructions being executed by the processor. This attack sequence can be arbitrarily long to avoid false positives, and the particular sequence is hard coded into the hardware. This foothold requires relatively few transistors. We implement it by including a small state machine in the integer unit (*i.e.*, the pipeline) of the processor that looks for the triggering sequence of instructions, and asserts the supervisor-mode bit when enabled.

7.2 Memory redirection foothold

Our memory redirection foothold provides hardware support for unprivileged malicious software by allowing an attacker to access arbitrary virtual memory locations.

This foothold uses a sequence of bytes as the trigger. In this case, when the foothold observes store instructions with a particular sequence of byte values it then interprets the subsequent bytes as the *redirection address*. The malicious logic records the address of the block and the redirection address in hardware registers. The next time the address is loaded from memory, the malicious hardware substitutes the redirection address as the address to be loaded and asserts the supervisor bit passed to the memory management unit (MMU). That is, the next read to this block will return the value of a different location in the memory. Memory writes are handled analogously, in that the next write to the block is redirected to write to the redirection address. The net effect is providing full access to arbitrary virtual memory locations and bypassing MMU protections enforced in the processor.

This foothold provides flexibility for attackers because attackers can trigger the circuit using only data values. Attackers can trigger the foothold by injecting specific data values into a system using a number of techniques including unsolicited network packets, emails, and images on web sites. By using these mechanisms to arbitrarily manipulate the system's memory, a remote attacker can compromise the system, for example, by searching memory for encryption keys, disabling authentication checks by modifying the memory of the targeted system, or altering executable code running on the system.

7.3 Shadow mode foothold

The shadow mode foothold allows an attacker to inject and execute arbitrary code. The shadow mode foothold works by monitoring data values as they pass between the cache and the pipeline, and installs an invisible firmware within the processor when a specific value triggers the attack. When this firmware runs, it runs with full processor privileges, it can gain control of the processor at any time, and it remains hidden from software running on the system. To provide storage for exploit instructions and data, this foothold reserves blocks in the instruction and data caches for storing injected instructions and data. The shadow mode foothold is triggered with a sequence of bytes and the shadow mode foothold interprets the bytes following the trigger sequence as commands and machine instructions.

To evaluate BlueChip, we implement the "bootstrap trigger" portion of the shadow mode foothold. The bootstrap trigger waits for a predetermined value to be stored to the data cache, and asserts a processor exception that transfers control to a hard-coded "bootstrap code" that resides within the processor cache. Our implementation includes the "bootstrap trigger" and asserts a processor exception, but omits the "bootstrap code" portion of the foothold. As a result, we are unable to implement full system attacks using our version of the foothold, but it does give us enough of the functionality of the shadow mode foothold to enable us to evaluate our defense because removing the "bootstrap trigger" disables the attack.

8 BlueChip prototype

To experimentally verify the BlueChip approach, we prototyped the hardware, the software, and the design-time UCI analysis algorithm.

We based our hardware implementation on the Leon3 processor [15] design. Our prototype is fully synthesizable and runs on an FPGA development board that includes a Virtex 5 FPGA, CompactFlash, Ethernet, USB, VGA, PS/2, and RS-232 ports. The Leon3 processor implements the SPARC v8 instruction set [24] and our configuration uses eight register windows, a 16 KB instruction cache, a 64 KB data cache, includes an MMU, and runs at 100 MHz, which is the maximum clock rate we are able to achieve for the unmodified Leon3 design, for our target FPGA. For the software, we use a SPARC port of the Linux 2.6.21.1 kernel on our FPGA board and we install a full Slackware distribution on our system. By evaluating BlueChip on an FPGA development board and by using commodity software, we have a realistic environment for evaluating our hardware modifications and accompanying software systems.

To insert our BlueChip hardware modifications, we wrote tools that take as input data-flow pairs generated by our UCI implementation and automatically transforms the Leon3 VHDL source code to replace suspicious circuits and add exception delivery logic. Our tool is mostly hardware-implementation agnostic and should work on a wide range of hardware designs automatically. The only hardware implementation specific portions of our tool are for connecting the BlueChip logic to the Leon3 exception handling stage.

For our UCI implementation, we wrote a VHDL compiler front end in Java which generates a data-flow graph from arbitrary HDL source code, determines all possible pairs of edges in the data-flow graph, then uses TCL to automatically drive ModelSim, running the simulation and removing pairs that are violated during testing. The last stage of our UCI implementation performs a source-to-source transformation using the original VHDL design and remaining data-flow pairs to generate a design with BlueChip hardware.

Our BlueChip software runs as an exception handler within the Linux kernel. The BlueChip emulation code is written in C and it can emulate all non-privileged SPARC instructions and most of the privileged operations of the Leon3 SPARC implementation. Because SPARC is a reduced instruction set computer (RISC), we implemented our emulator using only 1759 lines of code and it took us about a week to implement our instruction emulation routines.

We identify suspicious hardware using three sets of tests: the basic test suite included with the Leon3 distribution, SPARC certification test cases from SPARC International, and five additional test cases to test portions of the instruction set specification that are uncovered by the basic Leon3 and SPARC certification test cases. To identify suspicious logic, we simulate the HDL using ModelSim version 6.5 and perform the UCI analysis on the simulation results. Our analysis focuses on the integer unit (*i.e.*, the core pipeline) of the Leon3 processor.

9 BlueChip evaluation

This section describes our evaluation of BlueChip. In our evaluation, we measure BlueChip's: (1) ability to stop attacks, (2) ability to successfully emulate instructions that used hardware removed by BlueChip, and (3) hardware and software overheads.

9.1 Methodology

To evaluate BlueChip's ability to prevent and recover from attacks, we wrote software that activates the malicious hardware described in Section 8. We designed the software to activate and exploit the low-level footholds implemented by our attacks, and tested to see if these out-of-spec abstractions were rendered powerless and if the system could make post attack progress.

To identify suspicious circuits, we used three different sets of hardware design verification tests. First, we used the Gaisler test suite that comes bundled with the Leon3 hardware's HDL code. These test cases use ISA-level instructions to test both the processor core and peripheral (*i.e.*, outside the processor core) units like the caches, memory management unit, and system-on-chip units such as the UART. Second, we used the official SPARC verification tests from SPARC International, which are used to ensure compatibility with the SPARC instruction set. These test cases are designed to confirm that a processor implements the instructions and architecturally visible states needed to be considered a SPARC processor, but they are not intended to be a complete design verification suite. Third, we created a small set of custom hardware test cases to improve design coverage, closer to what is common in a production environment. The custom test cases cover gaps in the Gaisler test cases and exercises instructions that Leon3 supports, but are optional in the SPARC ISA specification (*e.g.*, floating-point operations).

To measure execution overhead, we used three workloads that stressed different parts of the system: `wget` fetches an HTML document from the Web and represents a network bound workload, `make` compiles portions of the `ntpdate` application and stresses the interaction between kernel and user modes, and `djpeg` decompresses a 1MB jpeg image as a representative of a compute-bound workload. To address variability in the measurements, reported execution time results are the average of 100 executions of each workload relative to an uninstrumented base hardware configuration. All of our overhead experiments have a 95% confidence interval of less than 1% of the average execution time.

Attack	Prevent	Recover
Privilege Escalation	√	√
Memory Redirection	√	
Shadow Mode	√	√

Figure 8: BlueChip attack prevention and recovery.

9.2 Does BlueChip prevent the attacks?

There are two goals for BlueChip when aiming to defend against malicious hardware. The first and most important goal is to prevent attacks from influencing the state of the system. The second goal is for the system to recover, allowing non-malicious programs to make progress after an attempted attack.

The results in Figure 8 show that BlueChip successfully prevents all three attacks, meeting the primary goal for success. BlueChip meets the secondary goal of recovery for two of the three attacks, but it fails to recover from attempted activations of the memory redirection attack. In this case, the attack is prevented, but software emulation is unable to make forward progress. Upon further examination, we found that the Leon3's built-in pipeline recovery mechanism was insufficient to clear the attack's internal state. This lack of progress is due to the attack circuit ignoring the Leon3 control signal that resets registers on pipeline flushes, thus making some attack states persist even after pipeline flushes. This situation causes the BlueChip hardware to repeatedly trap to software, thus blocking forward progress, but preventing the attack. Our analysis indicates that augmenting Leon3's existing recovery mechanism to provide additional state recovery would allow BlueChip to recover from this attack as well.

9.3 Is software emulation successful?

BlueChip justifies its aggressive identification and removal of suspicious circuits by relying on software to emulate any mistakenly removed functionality. Thus, BlueChip will trigger spurious exceptions (*i.e.*, those exceptions that result from removal of logic mistakenly identified as malicious). In our experiments, all of the benchmarks execute correctly, indicating BlueChip correctly recovers from the spurious BlueChip exceptions that occurred in these workloads.

Figure 9 shows the average rate of BlueChip exceptions for each benchmark. Even in the worst case, where a BlueChip exception occurs every 20ms on average, the frequency is far less than the operating system's timer interrupt frequency. The rate of BlueChip exceptions is low enough to allow for complex software handlers without sacrificing performance.

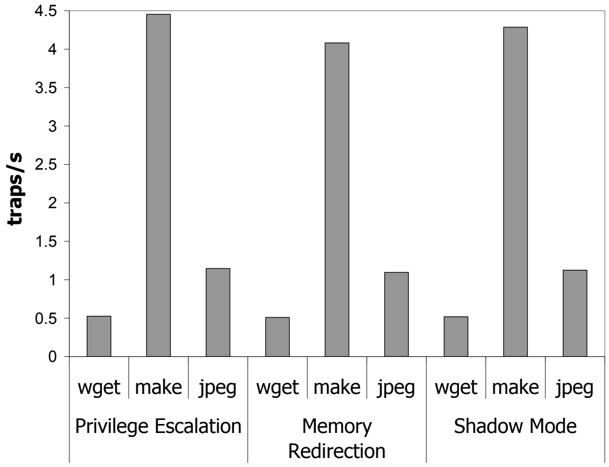


Figure 9: BlueChip software invocation frequencies.

Figure 10 shows the experimental data used to quantify the effectiveness of UCI. This figure shows the number of suspicious pairs remaining after each stage of testing. The misidentified pairs, which are all false positives for our tests, are the number of suspicious pairs minus the number of attack pairs detected. These false positive pairs can manifest themselves as spurious BlueChip exceptions during runtime. The number of pairs remaining after testing affects the likelihood of seeing spurious BlueChip exceptions, with fewer pairs generally leading to less frequent traps. Even though some of the remaining pairs result in spurious exceptions, the instruction-level emulation provided by BlueChip software hides this behavior from the rest of the system, thus allowing unmodified applications to execute unaware that they are running on BlueChip hardware.

The discrepancy in the number of traps experienced by each benchmark is also worth noting. The `make` benchmark experiences the most traps, by almost an order of magnitude. Looking at the UCI pairs that fire during testing, and looking at the type of workload `make` creates, the higher rate of traps comes from interactions between user and kernel modes. This happens more often in `make` than the other benchmarks, as `make` creates a new process for each compilation. More in-depth tracing of the remaining UCI pairs reveals that many pairs surround the interaction between kernel mode and user mode. Because UCI is inherently based on design verification tests, this perhaps indicates the parts of hardware least tested in our three test suites. Conversely, the relatively small rate of BlueChip exceptions experienced by `wget` is due to its I/O (network) bound workload. Most of the time is spent waiting for packets, which apparently does not violate any of the UCI pairs remaining after testing.

	Baseline		Gaisler Tests		+SPARC Tests		+Custom Tests	
	Attack	**All**	**Attack**	**All**	**Attack**	**All**	**Attack**	**All**
Privilege Escalation	2	3046	1	103	1	87	1	39
Memory Redirection	54	3098	8	110	8	94	8	46
Shadow Mode	8	3051	1	103	1	87	1	39

Figure 10: UCI dataflow pairs. This figure shows how many dataflow pairs UCI identifies as suspicious for the Gaisler test suite, the SPARC verification test suite, and our custom test cases, cumulatively. The data shows the number of dataflow pairs identified total, and shows how many of these pairs are from attack circuits.

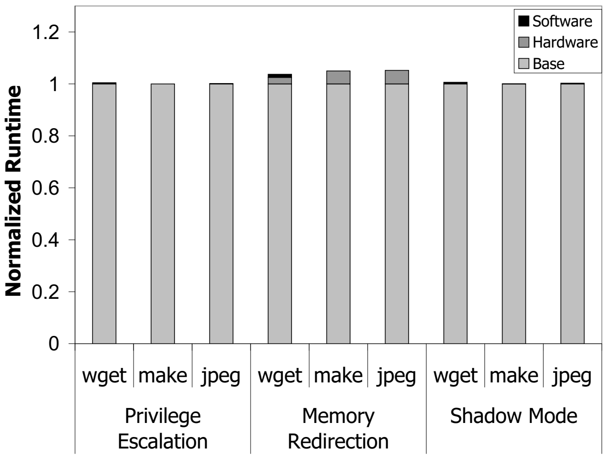


Figure 11: Application runtime overheads for BlueChip systems.

Attack	Power (W)	Area (Luts)	Freq. (MHz)
Privilege Escalation	0.41%	1.38%	0.00%
Memory Redirection	0.47%	1.19%	5.00%
Shadow Mode	0.29%	0.31%	0.00%

Figure 12: BlueChip hardware overheads for each of our attacks.

9.4 Is BlueChip's runtime overhead low?

Although BlueChip successfully executes our benchmark workloads, frequent spurious exceptions have the potential to significantly impact system performance. Furthermore, BlueChip's hardware transformation could impact the hardware's operating frequency.

Figure 11 shows the normalized breakdown of runtime overhead experienced by the benchmarks running on a BlueChip system versus an unprotected system. The runtime overhead from the software portion of BlueChip is just 0.3% on average. The software overhead comes from handling spurious BlueChip exceptions, primarily from just two of the UCI pairs. The average overhead from the hardware portions of BlueChip is approximately 1.4%.

Figure 12 shows the relative cost of BlueChip in terms of power, device area, and maximum operating frequency. The hardware overhead in terms of area averages less than 1% of the entire design. Even though BlueChip needs hardware to monitor the remaining pairs, much of the hardware already exists and BlueChip just taps the pre-existing signals. The majority of the area overhead comes from the comparisons used to determine if a BlueChip exception is required given the state of the pipeline. To reduce BlueChip's impact on maximum frequency, these comparisons happen in parallel and BlueChip's exception generation uses a tree of logical-OR operations. In fact, for the privilege escalation and shadow mode versions of BlueChip, there is no measurable impact on maximum frequency, indicating that UCI pair checking hardware is typically off the critical path. For the memory redirection attack hardware design, some of the pair checking logic is placed on the memory path, which is the critical path for this design and target device. In this case, the maximum frequency is reduced by five percent. Consistent with the small amount of additional hardware, the power overhead averages less than 0.5%.

10 Additional related work

In addition to specific work discussed in previous sections, our work is more broadly related to research on attacks and defenses. We are not the first to look at malicious hardware and defenses. However, prior work has focused on other aspects of malicious hardware, such as fabrication-level supply chain attacks, detection of counterfeit chips, programmable hardware [17] and isolation of circuits within FPGAs. This section describes work on detecting and isolating malicious hardware, detecting source-code-level security flaws, and hardware extensibility.

10.1 Trojan detection & isolation

Agrawal *et al.* [3] propose signal processing techniques to detect additional circuits through power side-channel power analysis. This promising approach faces two key challenges. First, it assumes that the defender has a reference copy of the chip without a trojan circuit, an assumption breaks down if an attacker can modify the design of the circuit directly. Second, the results are for power simulations on small (on the order of 1000 gate) circuits. Although the results are promising, it is unclear how well these techniques will work in practice and on large circuits, such as microprocessors.

Huffmire *et al.* [19] propose isolation primitives for hardware components designed to run on field programmable gate array (FPGA) hardware. These isolation primitives can help system builders reason about connections between components, but provide little or no protection against potentially malicious central components — such as a memory controller or processor — that need to communicate with most or all components on the system.

Process variations cause each chip to behave slightly differently, and such analog side effects can be used to identify individual chips. Gassend, *et al.* [16] use this fact to create physically random functions (PUFs) that can be used to uniquely identify individual chips. Chip identification ensures that chips are not swapped in transit, but provides no detection capabilities for chips that include malicious hardware in the design.

10.2 Detecting source code modifications

Formal methods such as symbolic execution [9, 10], model checking [6, 13, 28], and information flow [23, 29] have been applied to software systems for better test coverage and improved security analysis. These diverse approaches can be viewed as alternatives to UCI, and may provide promising extensions for UCI to detect malicious hardware if correct abstractions can be developed.

10.3 Hardware extensibility

In some respects, the BlueChip system resembles previous work on hardware extensibility, where designers use various forms of software to extend and change the way deployed hardware works. Some examples of this type of hardware extensibility include patchable microcode [18], firmware-based TLB control in Itanium processors [2], Transmeta code morphing software [14], and Alpha PAL code [5]. Our design uses some of the same hardware mechanisms used in these systems, but for coping with malicious hardware rather than design bugs.

11 Conclusions

BlueChip neutralizes malicious hardware introduced at design time by identifying and removing suspicious hardware during the design verification phase, while using software at runtime to emulate hardware instructions to avoid erroneously removed circuitry.

Experiments indicate that BlueChip is successful at identifying and preventing attacks while allowing non-malicious executions to make progress. Our malicious circuit identification algorithm, UCI, relies on the attempts to hide functionality to identify candidate circuits for removal. BlueChip replaces circuits identified by UCI with exception logic, which initiates a trap to software. The BlueChip software emulates instructions to detour around the removed hardware, allowing the system to attempt to make forward progress. Measurements taken with the attacks inserted show that such exceptions are infrequent when running a commodity operating system using traditional applications.

In summary, these results show that addressing the malicious insider problem for hardware design is both possible and worthwhile, and that approaches can be cost effective and practical.

Acknowledgments

The authors thank David Wagner, Cynthia Sturton, Bob Colwell, and the anonymous reviewers for comments on this work. We thank Jiri Gaisler for assistance with the Leon3 processor and Morgan Slain and Chris Larsen from SPARC International for the SPARC certification test benches.

This research was funded in part by the National Science Foundation under grants CCF-0811268, CCF-0810947, CCF-0644197, and CNS-0953014, AFOSR MURI grant FA9550-09-01-0539, ONR under N00014-09-1-0770 and N00014-07-1-0907, donations from Intel Corporation, and a grant from the Internet Services Research Center (ISRC) of Microsoft Research. Any opinions, findings and conclusions or recommendations expressed in this paper are solely those of the authors.

References

[1] Maxtor basics personal storage 3200. `http://www.seagate.com/www/en-us/support/downloads/personal_storage/ps3200-sw`.

[2] Intel Itanium Processor Firware Specifications. `http://www.intel.com/design/itanium/firmware.htm`, March 2010.

[3] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan detection using IC fingerprinting. In

Proceedings of the 2007 IEEE Symposium on Security and Privacy, May 2007.

[4] Apple Computer Inc. Small Number of Video iPods Shipped With Windows Virus. 2006. `http://www.apple.com/support/windowsvirus/`.

[5] P. Bannon and J. Keller. Internal architecture of Alpha 21164 microprocessor. *compcon*, 00:79, 1995.

[6] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. ACM/IEEE Annual Design Automation Conference*, pages 317–320, 1999.

[7] Bob Colwell. Personal communication, March 2009.

[8] BugTraq Mailing List. Irssi 0.8.4 backdoor, May 2002. `http://archives.neohapsis.com/archives/bugtraq/2002-05/0222.html`.

[9] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 2008 USENIX Symposium on Operating Systems Design and Implementation*, December 2008.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 322–335, New York, NY, USA, 2006. ACM.

[11] CERT. Cert advisory ca-2002-24 trojan horse openssh distribution. Technical report, CERT Coordination Center, 2002. `http://www.cert.org/advisories/CA-2002-24.html`.

[12] CERT. Cert advisory ca-2002-28 trojan horse sendmail distribution. Technical report, CERT Coordination Center, 2002. `http://www.cert.org/advisories/CA-2002-28.html`.

[13] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, pages 171–185, 2004.

[14] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. K. A. Klaiber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adapative retranslation to address real-life challenges. In *Proceedings of the 1st IEEE/ACM Symposium of Code Generation and Optimization*, Mar 2003.

[15] Gaisler Research. Leon3 synthesizable processor. http://www.gaisler.com.

[16] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 148–160, New York, NY, USA, 2002. ACM Press.

[17] I. Hadžić, S. Udani, and J. M. Smith. FPGA Viruses. In *Proceedings of 9th International Workshop on Field-Programmable Logic and Applications, FPL'99*, LNCS. Springer, August 1999.

[18] L. C. Heller and M. S. Farrell. Millicode in an IBM zSeries processor. *IBM J. Res. Dev.*, 48(3-4):425–434, 2004.

[19] T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine. Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 281–295, Washington, DC, USA, 2007. IEEE Computer Society.

[20] P. A. Karger and R. R. Schell. Multics Security Evaluation: Vulnerability Analysis. Technical Report ESD-TR-74-192, HQ Electronic Systems Division: Hanscom AFB, June 1974.

[21] P. A. Karger and R. R. Schell. Thirty Years Later: Lessons from the Multics Security Evaluation. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference*, page 119. IEEE Computer Society, 2002.

[22] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and implementing malicious hardware. In *Proceedings of the First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, April 2008.

[23] A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *Proceedings of the 1997 Symposium on Operating Systems Principles*, October 1997.

[24] SPARC International Inc. SPARC v8 processor. http://www.sparc.org.

[25] B. Sullivan. Digital picture frames infected with virus, January 2008. `http://redtape.msnbc.com/2008/01/digital-picture.html`.

[26] K. Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, 1984.

[27] United States Department of Defense. Mission impact of foreign influence on DoD software. September 2007.

[28] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *OSDI'04: Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, pages 273–288, Berkeley, CA, USA, 2004. USENIX Association.

[29] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: secure program partitioning. In *Proceedings of the 2001 Symposium on Operating Systems Principles*, October 2001.