MATTHEW HICKS, MURPH FINNICUM, SAMUEL T. KING, MILO M.K. MARTIN, AND JONATHAN M. SMITH

overcoming an untrusted computing base: detecting and removing malicious hardware automatically

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 article presents Unused Circuit Identification (UCI), an approach for detecting suspicious circuits during design time, and BlueChip, a hybrid hardware/software approach to detaching suspicious circuits and making up for UCI classifier errors during runtime.

Matthew Hicks is a fifth-year graduate student in computer science at the University of Illinois. His current research focuses on the border between hardware and system software, with a focus on FPGAs and specialized operating systems.

mdhicks2@illinois.edu

Murph Finnicum is a graduate student in computer science at the University of Illinois, where he also completed his undergraduate studies in computer engineering. His current research includes work on ambiguity-tolerant automatic programming and novel techniques to make Web browsers faster.

mfinnic2@illinois.edu

Sam King is an assistant professor in the Computer Science Department at the University of Illinois. His primary research interests are in security, operating systems, and computer architecture.

kingst@illinois.edu

Milo Martin is an associate professor in the Computer and Information Science Department at the University of Pennsylvania. His research focuses on making computers easier to design, verify, and program. Specific projects include transactional memory, adaptive cache coherence protocols, hardware-aware verification of concurrent software, and hardware-assisted memory-safe implementations of the C programming language. Dr. Martin is a recipient of the NSF CAREER award and received a PhD from the University of Wisconsin—Madison.

milom@cis.upenn.edu

Jonathan M. Smith is the Olga and Alberico Pompa Professor of Engineering and Applied Science and a professor of computer and information science at the University of Pennsylvania. He served as a program manager at DARPA 2004–2006 and was awarded the OSD Medal for Exceptional Public Service in 2006. He is an IEEE Fellow. His current research interests range from programmable network infrastructures and cognitive radios to disinformation theory and architectures for computer augmented immune response.

jms@cis.upenn.edu

Modern hardware design processes in many ways 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 [11].

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 to recall five million chips) has been estimated to cost many billions of dollars today [3]. 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 that was inserted during design time. For example, attackers might introduce a circuit that detects a certain 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.

During the design phase, UCI flags as suspicious any unused circuitry (any circuit not activated by any of the many design verification tests). BlueChip disconnects these suspicious circuits from the rest of the trusted circuit. 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 BlueChip exception handler software is responsible for emulating the overall behavior of the hardware to allow the system to make forward progress. 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.

Motivation and Attack Model

This article 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 [14], system designers inserting malicious source code intentionally [4, 9, 10], compromised servers that host modified source code [5, 6], and products that come pre-installed with malware [1, 2, 13]. 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 [15].

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 insert malicious circuits into a design prior to final design validation with the hope that the changes will evade detection. The malicious hardware demonstrated in previous work [11] supports the plausibility of this scenario, in that small and localized changes (e.g., tens of lines in 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 attack the manufacturing or distribution supply chain. We assume that only one or a few individuals act 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.

UCI and BlueChip can be used by anyone from designers to debuggers, but the target audience is the lead designer or system integrator who advances the design to the fabrications stage. Our work assumes that this person is trustworthy and has much to lose if the hardware contains malicious circuitry. This work also relies on a testing regimen based on simulation at the hardware description level. Here the designer with signoff responsibilities can view any wire in the design, during any given cycle, and has the ability to add or remove test cases. We assume that no extra rigging is required for specialized testing (e.g., boundary scan chains); what is simulated is what will be in the fabricated chip.

The BlueChip Approach

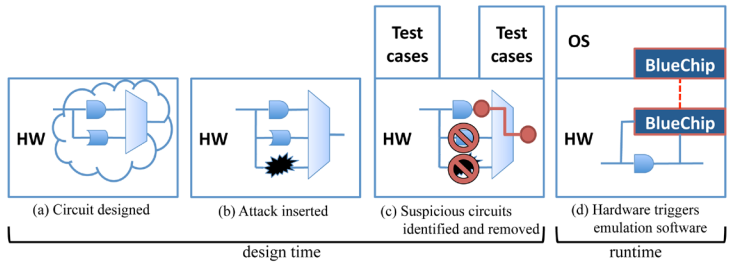


FIGURE 1: OVERALL BLUECHIP ARCHITECTURE. THIS FIGURE SHOWS THE OVERALL FLOW FOR BLUECHIP WHERE (A) DESIGNERS DEVELOP HARDWARE DESIGNS AND (B) A ROGUE DESIGNER INSERTS MALICIOUS LOGIC INTO THE DESIGN. DURING DESIGN VERIFICATION PHASE, (C) BLUECHIP IDENTIFIES AND REMOVES SUSPICIOUS CIRCUITS AND INSERTS 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.

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

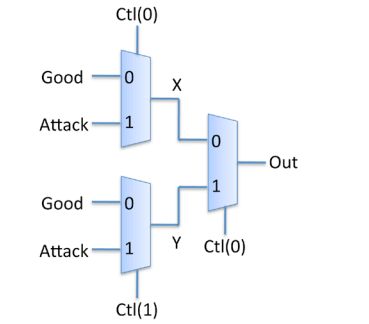
Overall, BlueChip provides a separation between the responsibilities of the hardware and the software. The BlueChip hardware prevents attacks by removing suspicious circuits. The BlueChip software ensures forward progress by emulating instructions. If an attacker is able to control the BlueChip software it does *not* give attackers any additional capabilities—the BlueChip hardware still neutralizes the attack—but usurping BlueChip software could prevent the system from making forward progress.

For more information about the design and implementation of our BlueChip hardware and software, please see our recent paper on the topic [8].

Detecting Suspicious Circuits

One key component of the overall system is the algorithm for detecting suspicious circuits. Our goal is to develop an algorithm that identifies all malicious circuits without identifying benign circuits. In addition, our technique should be impossible 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 and system integrators 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.




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

FIGURE 2: CIRCUIT DIAGRAM AND HDL SOURCE CODE FOR A MULTIPLEXOR (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. GOOD, ATTACK, X, Y, AND OUT ARE ALL WIRES WITH ARBITRARY VALUES THAT ARE FREE TO CHANGE DURING SIMULATION. FOR SIMPLICITY, ASSUME GOOD AND ATTACK NEVER CARRY THE SAME VALUE. WIRES GOOD AND ATTACK ARE THE INPUTS AND WIRE OUT IS THE OUTPUT. THE WIRES LABELED CTL(X) ARE THE CONTROL LINES FOR THEIR RESPECTIVE MUXES. THE VALUE ON THE CONTROL LINE OF A MUX DETERMINES WHICH INPUT GETS ITS VALUE PASSED ALONG AS THE OUTPUT OF THE MUX. THE 0 AND 1 LABELS ON EACH MUX IN THIS FIGURE SHOW WHICH CONTROL VALUE DRIVES WHICH INPUT VALUE TO THE OUTPUT.

An attacker can easily craft circuits that yield 100% code coverage after testing, but test cases never actually trigger the attack. For example, Figure 2 shows a multiplexer (mux) circuit that has 100% code coverage without outputting the attack value. If the verification test suite includes control states (value of "Ctl(0,1)") 00, 01, and 10, all lines of code that make up the circuit will be covered, but the output value on wire "Out" will always be the same value as the value on wire "Good." We apply this evasion technique to the attacks we evaluate and find that it does evade code coverage–based detection.

Our approach is to use design verification tests to help detect malicious circuits, repurposing functional verification tests as security verification tests. If an attack circuit contaminates the output for a test case, the designer would know that the circuit is operating out-of-spec, 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 [11]. 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 space of a design, but can still enable attacks in the field by inducing the trigger sequence. 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. Our proposal is to consider circuits suspicious whenever a design includes them, but the circuit does *not* affect any of the outputs for any of the test cases.

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.

```
// 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 for each simulation clock cycle
   for each pair in pairs
      if the sink and source not equal
        remove the pair from the pairs set
```

FIGURE 3: IDENTIFYING POTENTIALLY MALICIOUS CIRCUITS USING OUR UCI ALGORITHM

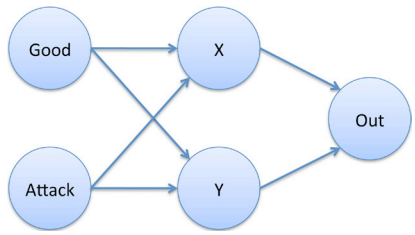


FIGURE 4: DATA-FLOW GRAPH FOR MUX REPLACEMENT CIRCUIT

To identify potentially malicious circuits, our algorithm performs two steps (Figure 3). First, UCI creates a data-flow graph for the design (Figure 4). In this graph, nodes are signals (wires) and state elements (to account for data flow across clock cycles); 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 direct dependencies (e.g., (Good, X) in Figure 4) and indirect dependencies (e.g., (Good, Out) in Figure 4). For more details on the third member of data-flow tuples, refer to our recent paper [8]. Each data-flow tuple effectively states that all the logic between the source and the sink wire can be replaced by a short-circuit-like delay line.

Second, UCI simulates the HDL code using standard design verification tests to find the set of data-flow tuples where intermediate logic does *not* affect the data value that flows between the source and sink wires. To test for this condition, at each simulation step UCI checks for inequality for each of the remaining data-flow tuples. If the current value of the sink is not equal to the value of the source DELAY cycles ago, this implies, conservatively, that the logic between the two wires has an effect on the value. UCI removes this tuple from suspicion, as there is now a case where the intermediate logic had an effect and the effect was verified within the design's specification. More clearly, for registers, UCI accounts for latched data by maintaining a history of simulation values, allowing it make the appropriate comparison of source and sink wire values when they are separated by state elements.

After the simulation completes, UCI has a set of remaining data-flow tuples where the logic in between the source and sink wires of the tuple does not affect the value as it travels, possibly across clock cycles, from source to

sink. In other words, we could replace the intermediate logic with a wire, possibly including some delay state elements, and it would not affect the overall behavior of the circuit, in any way, for any of the design verification tests.

Consider how this algorithm works for the mux-replacement circuit shown in Figure 2, when the attack lies dormant during the test cases (values of "Attack" and "Out" are not equal):

- UCI creates the initial set of data-flow tuples, (Good, X, 0), (Attack, X, 0), (Good, Y, 0), (Attack, Y, 0), (Good, Out, 0), (Attack, Out, 0), (X, Out, 0), and (Y, Out, 0).
- UCI considers the first simulation step where Ctl(0) and Ctl(1) are 0. Thus, X, Y, and Out all carry the same value as Good. UCI removes tuples (Attack, X, 0), (Attack, Y, 0), and (Attack, Out, 0), since X, Y, and Out don't carry the same value as Attack.
- UCI considers the second simulation step where Ctl(0) is 0 and Ctl(1) is 1. Thus, X and Out carry the same value as Good, while Y carries the same value as Attack. UCI removes tuples (Good, Y, 0) and (Y, Out, 0), since Y doesn't carry the same value as Good or Out.
- UCI considers the third simulation step where Ctl(0) is 1 and Ctl(1) is 0. Thus, X carries the same value as Attack while Y and Out carry the same value as Good. UCI removes tuples (Good, X, 0) and (X, Out, 0) since X doesn't carry the same values as Good or Out.
- UCI finishes the simulation, and the only remaining tuple is (Good, Out, 0). This tells the designer that the intermediate logic between wires Good and Out doesn't have any real effect for the test cases. This implies that wires Good and Out can be short-circuited with no adverse effects.

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 wire labeled "Good" directly to the wire labeled "Out," thus removing the malicious logic from the design.

BlueChip Evaluation

In our evaluation, we measure (1) BlueChip's ability to stop attacks, (2) BlueChip's ability to successfully emulate instructions that used hardware removed by BlueChip, and (3) the runtime overhead of our system.

We based our hardware implementation on the Leon3 processor [7] 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 [12], and our configuration uses eight register windows, a 16KB instruction cache, and a 64KB data cache, includes an MMU, and runs at 100MHz, 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 evaluate BlueChip's ability to prevent and recover from attacks, we wrote software that activates the malicious hardware described in our recent papers [11, 8]. Our prior work on developing hardware attacks focused on adding minimal additional logic gates as a *foothold* for a system-level attack. We explored three such footholds: the *supervisor transition* foothold

enables an attacker to transition the processor into supervisor mode to escalate the privileges of user-mode code, the *memory redirection* foothold enables an attacker to read and write arbitrary virtual memory locations, and 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 log in to a remote system automatically [11].

To identify suspicious circuits, we used the Gaisler test suite that comes bundled with the Leon3 hardware's HDL code, the official SPARC verification tests from SPARC International, and a few custom test cases we wrote for this experiment.

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.

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.

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

FIGURE 5: BLUECHIP ATTACK PREVENTION AND RECOVERY

The results in Figure 5 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 attack stored state that fell outside of our BlueChip hardware mechanisms. We believe that this limitation is an artifact of our current implementation and could be fixed by using a more sophisticated hardware analysis algorithm.

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 that BlueChip correctly recovers from the spurious BlueChip exceptions that occurred in these workloads.

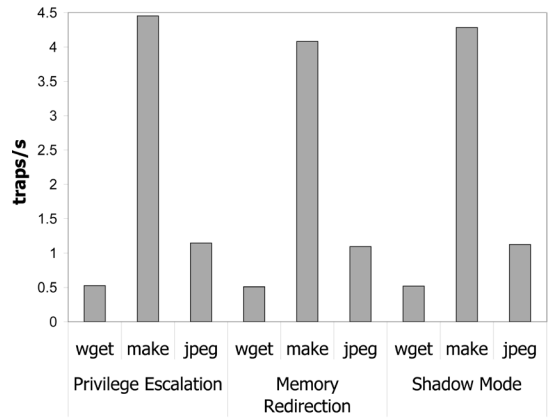


FIGURE 6: BLUECHIP SOFTWARE INVOCATION FREQUENCIES

Figure 6 shows the average rate of BlueChip exceptions for each benchmark. Even in the worst case, where a BlueChip exception occurs every 200ms 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.

The discrepancy in the number of traps experienced by each benchmark is 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.

IS BLUECHIP'S RUNTIME OVERHEAD LOW?

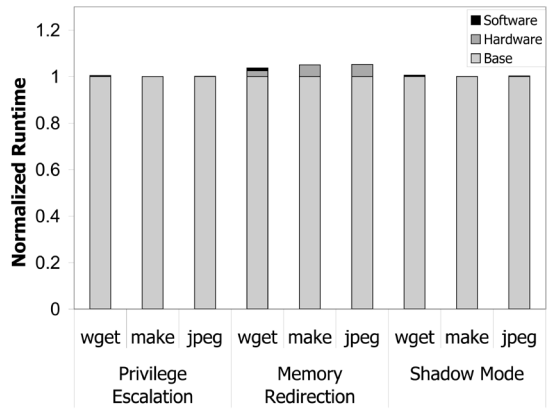


FIGURE 7: APPLICATION RUNTIME OVERHEADS FOR BLUECHIP SYSTEMS

Although BlueChip successfully executes our benchmark workloads, frequent spurious exceptions have the potential to significantly impact system performance.

Figure 7 shows the normalized breakdown of runtime overhead experienced by the benchmarks running on a BlueChipped 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, including the cases with zero hardware overhead, is approximately 1.4%.

Conclusion

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://seagate.custkb.com/seagate/crm/selfservice/search.jsp?DocId=205131&NewLang=en.

[2] Apple Computer Inc., Small Number of Video iPods Shipped with Windows Virus. 2006: http://www.apple.com/support/windowsvirus/.

[3] Bob Colwell, personal communication, March 2009.

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

[5] CERT Advisory CA-2002-24 Trojan Horse OpenSSH Distribution, technical report, CERT Coordination Center, 2002: http://www.cert.org/advisories/CA-2002-24.html.

[6] CERT Advisory CA -2002-28 Trojan Horse Sendmail Distribution, technical report, CERT Coordination Center, 2002: http://www.cert.org/advisories/CA-2002-28.html.

[7] Gaisler Research, Leon3 synthesizable processor: http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53.

[8] 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 *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, May 2010.

[9] 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.

[10] 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* (IEEE Computer Society, 2002), p. 119.

[11] 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 '08)*, April 2008.

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

[13] B. Sullivan, "Digital Picture Frames Infected with Virus," January 2008: http://redtape.msnbc.com/2008/01/digital-picture.html.

[14] K. Thompson, "Reflections on Trusting Trust," *Communications of the ACM*, vol. 27, no. 8, 1984, pp. 761–63.

[15] United States Department of Defense, "Mission Impact of Foreign Influence on DoD Software," September 2007.