Designing and implementing malicious hardware

Samuel T. King, Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou
University of Illinois at Urbana Champaign, Urbana, IL 61801

Abstract

Hidden malicious circuits provide an attacker with a stealthy attack vector. As they occupy a layer below the entire software stack, malicious circuits can bypass traditional defensive techniques. Yet current work on trojan circuits considers only simple attacks against the hardware itself, and straightforward defenses. More complex designs that attack the software are unexplored, as are the countermeasures an attacker may take to bypass proposed defenses.

We present the design and implementation of Illinois Malicious Processors (IMPs). There is a substantial design space in malicious circuitry; we show that an attacker, rather than designing one specific attack, can instead design hardware to support attacks. Such flexible hardware allows powerful, general purpose attacks, while remaining surprisingly low in the amount of additional hardware. We show two such hardware designs, and implement them in a real system. Further, we show three powerful attacks using this hardware, including a login backdoor that gives an attacker complete and high-level access to the machine. This login attack requires only 1341 additional gates: gates that can be used for other attacks as well. Malicious processors are more practical, more flexible, and harder to detect than an initial analysis would suggest.

1 Introduction

1.1 Motivation

Attackers may be able to insert covertly circuitry into integrated circuits (ICs) used in today's computer-based systems; a recent Department of Defense report [16] identifies several current trends that contribute to this threat. First, it has become economically infeasible to procure high performance ICs other than through commercial suppliers. Second, these commercial suppliers are increasingly moving the design, manufacturing, and testing stages of IC production to a diverse set of countries, making securing the IC supply chain infeasible. Together, commercial-off-the-shelf (COTS) procurement and global production lead to an "enormous and increasing" opportunity for attack [16].

Maliciously modified devices are already a reality. In 2006, Apple shipped iPods infected with the RavMonE virus [4]. During the cold war, the CIA sabotaged oil pipeline control software, which was then allowed to be "stolen" by Russian spies [10]. Conversely, Russian agents intercepted and modified typewriters which were to be used at the US embassy in Moscow; the modifications allowed the Russians to copy any documents typed on said typewriters [16]. Recently, external hard drives sold by Seagate in Taiwan were shipped with a trojan installed that sent personal data to a remote attacker [1]. Although none of these attacks use malicious circuits, they clearly show the feasibility of covertly inserting malicious elements in the COTS supply chain.

Using modified hardware provides attackers with a fundamental advantage compared to software-based attacks. Due to the lower level of control offered, attackers can more easily avoid detection and prevention. The recent SubVirt project shows how to use virtual-machine monitors to gain control over the operating system (OS) [11]. This lower level of control makes defending against the attack far more difficult, as the attacker has control over all of the software stack above. There is no layer below the hardware, thus giving such an attack a fundamental advantage over the defense.

Although some initial work has been done on this problem in the security community, our understanding of malicious circuits is limited. IBM developed a "trojan circuit" to steal encryption keys [3]. By selectively disabling portions of an encryption circuit they cause the encryption key to be leaked. This is the best example of an attack implemented in hardware that we are aware of, yet it has several shortcomings. First, their attack operates on hardware-level abstractions directly. They leak encryption keys from an encryption circuit and they ignore higher-level abstractions and system-level aspects. This limitation is problematic because security-critical information rarely maps directly to hardware-level abstractions. Second, although they describe and evaluate a defensive strategy, existing counter-strategies an attacker may employ are ignored [17, 18]. Finally, the attack itself is hard-coded; their malicious circuit is useful for only this one specific purpose.

Indeed, a single hard-coded attack in hardware greatly understates the power of malicious circuitry. This style of attack is an attack designed in hardware; *nobody has designed hardware to support attacks.* The design space of malicious circuitry is unexplored, outside of simple, special-purpose, hard-coded attacks. Responding to the threat of trojan circuits requires considering a variety of possible malicious designs; further, it requires anticipating and considering the attacker's counter-moves against our defenses. Without such consideration, we remain open to attack by malicious circuits.

1.2 Our contribution

In this paper, we address these concerns by designing and implementing Illinois Malicious Processors (IMPs). We show that IMPs are capable of powerful attacks, use surprisingly little circuitry, and are difficult to defend against. Our two primary contributions are:

We consider the malicious circuit design space. Beyond simply designing an attack in hardware, we consider hardware mechanisms to support general attacks. We design two malicious modifications of a CPU; a *memory access* mechanism and a *shadow mode* mechanism. The memory access mechanism allows an attacker to violate the target OS's isolation expectations, while shadow mode allows the hidden execution of an evil "firmware". Both attacks use a minimal number of transistors, yet allow the attacker wide access to a compromised system.

We design and implement potential attacks. Using our two malicious circuit designs, we design and implement three attacks. With the memory access mechanism, we design a privilege escalation attack, which gives an attacker root without checking credentials or creating log entries. Within shadow mode, we implement a login backdoor that lets an attacker log in as root without supplying a password, and we implement a service that steals passwords and sends them to the attacker. These attacks show the flexible nature of our attack circuitry. Further, despite the low level of our malicious circuitry, these attacks cause high level effects on the overall system.

To evaluate our ideas, we implement our IMPs both in simulation and in physical hardware. We modified the VHDL source of the Leon3 [8] processor (an open source SPARC design) to include malicious circuitry, and synthesized it to an embedded system development board. The resulting full system includes common components such as Ethernet, USB, VGA, etc., and is capable of running a complete Linux environment. Against this system, we carried out our login backdoor attack, and measured the real-world perturbation on the system.

We contribute in several ways. We are the first to design and implement general purpose hardware (two designs) to support the design of security attacks (three attacks). We show some of the design tradeoffs attackers may make when designing malicious circuitry, and the challenges attackers may face in implementing practical attacks with such circuits.

2 Problem statement, assumptions, and threat model

In this section we define the problem we address, state our assumptions, and describe our threat model.

We address the problem of designing and implementing malicious processors that carry out high-level attacks. In this paper we focus on an attacker that adds additional circuits to carry out the attack. We consider analog circuit perturbations (both timing and power), as well as discrete perturbations. We do not consider attacks where the gate-level design is unmodified and the attacker uses physical phenomena (e.g., excessive heat) to perturb execution or degrade performance of the circuit.

There are multitude of opportunities to insert hardware-based attacks, including the design, fabrication, packaging, testing, and integration stages (e.g., at a PC assembly plant).

Motivated attackers will subvert the IC supply chain if doing so provides sufficient value. Since modifying an IC is an expensive attack, it is doubtful that "script kiddies" will turn their adolescent energies to malicious processors, but the same cannot be said for attackers with resources. If malicious processors are capable of running valuable attacks, governments, organized crime syndicates, terrorist organizations, and so on will deploy them despite their cost. Historically, these types of organizations are experienced at covert operations, and have demonstrated considerable ingenuity in pursuing their goals. In contrast, there is little work on malicious processors.

3 Hardware designs

Previous work [3] presents a simple, pure hardware design space for attacks. Specifically, they attack a public-key encryption circuit. By turning off portions of the circuit a certain number of cycles in, they enable a key-leaking attack. Using a 16-bit counter for timing, their simulated malicious circuit takes an area of 406 gates. One can easily imagine other pure-hardware attacks, such as causing a circuit to fail or give incorrect results after some triggering condition is reached. Further, unless the output of the circuit is carefully monitored by duplicating the effort, the attack is difficult to detect. However, such special-purpose attacks are limited. The key leaking attack is only possible because the hardware they attack is cryptographic hardware. The only thing such a malicious circuit can be used for is stealing RSA encryption keys; if one wishes to steal AES keys, or steal plain text passwords, separate circuits must be designed. Further, while simple attacks are easy (e.g., fail after 2 years of operation), it is unclear how to realize semantically richer attacks (e.g., execute the SQL query `'DROP TABLE *;'`) using this technique.

Instead, we consider designing hardware mechanisms to enable malicious payloads. Specifically, we consider two mechanisms: a *memory access* mechanism that provides unchecked memory accesses and allows an attacker to bypass the protection of the memory management unit (MMU), and a *shadow mode* mechanism that allows attackers to execute an invisible malicious firmware. These two mechanisms represent different tradeoff points between analog perturbations, timing perturbations, and visibility from within the system, as well as the flexibility of the attack.

3.1 Memory access

Our *memory access* mechanism provides hardware support for unprivileged malicious software by allowing access to privileged memory regions. Malicious software triggers the attack by forcing a sequence of bytes on the data bus to enable the memory access circuits. This sequence can be arbitrarily long to avoid false positives, and the particular sequence must be agreed upon before deployment. Once the sequence is observed, the MMU in the data cache ignores CPU privilege levels for memory accesses, thus granting unprivileged software access to all memory, including privileged memory regions like the operating system's internal memory. In other words, loading a magic value on the data bus will disable protection checking. We implement this technique by modifying the data cache of our processor to include a small state machine that looks for the special sequence of bytes, plus some additional logic in the MMU to ignore privilege levels when malicious software enables the attack.

This mechanism requires relatively few transistors and is flexible; the attacker can use software to implement any payload they wish. Although the area of such an attack may be larger than a single, special built circuit, we can use the circuit for much more. For example, consider how large a special-built circuit would have to be to accept external control, execute arbitrary database queries, send the results off-site, and modify the logs to implicate another attacker. In comparison, the memory access mechanism is tiny. Unfortunately, the software of the attack is visible from within the system (at least until the attacker bootstraps more traditional rootkit mechanisms). Furthermore, the attacker must get their malicious software running on the system in the first place, albeit with any privilege level.

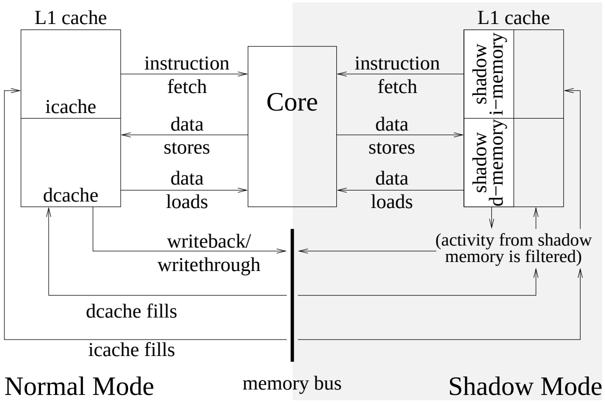


Figure 1: Hardware differences when shadow mode is active.

Using the memory access mechanism, we implement a privilege escalation program that bypasses the usual security checks (see Section 4.1).

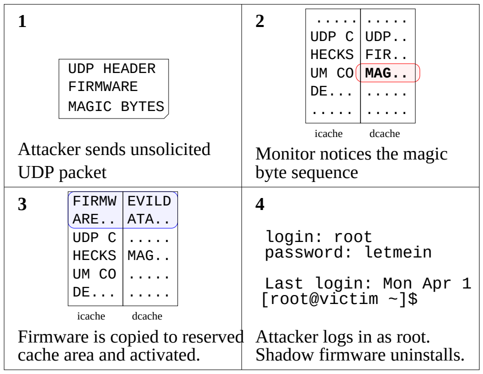
3.2 Shadow mode

Our *shadow mode* mechanism lies in-between pure hardware and pure software; we attempt to minimize the number of additional circuits needed to carry out the attack, remain hidden, and still support nearly arbitrary attacks. To minimize the number of additional circuits needed for an attack we reuse existing circuits by executing invisible instructions in a new processor mode called *shadow mode.* Shadow mode is similar to ISA extension modes, like Alpha PAL code [5] and Intel system management mode (SMM) [2], because shadow-mode instructions have full processor privileges and are invisible to software. However, unlike PAL code and SMM, we further aim to be invisible to hardware outside of our IMP. To hide attack instructions and data from hardware outside of our IMP, we reserve instruction cache lines and data cache lines for our attack, thus avoiding off-chip resources and preventing exposure of attack states and events that could be detected. However, reserving cache for the attack does perturb the timing of software running on the system. We quantify the impact of reserving cache lines for shadow mode in Section 6.

To load shadow-mode attacks on our IMP, we include two bootstrap mechanisms. One is to include a small section of *bootstrap code* that initializes the attack. We initialize the bootstrap code in the cache memory using the existing reset hardware (rather than using all zeros), or alternatively using SRAM reset bias mechanisms such as described in a recent patent [15]. This code consists of normal CPU instructions that are executed after a processor reset. While this code can then install an attack directly, more flexibly, the IMP will wait for a predetermined *bootstrap trigger*; a set of conditions to tell the IMP to load in a firmware from nearby data. The exact mechanism used to bootstrap attacks depends on the goals of the attacker, and the assumptions the IMP architect makes about the target platform. For example, many of our attacks assume the target platform includes a network interface. When a network interface is present an attacker can force data into a system easily by sending it an unsolicited network packet that the OS drops. For the OS to drop a network packet it must first inspect it, and the act of inspecting the network packet gives our bootstrap mechanism sufficient opportunity to look for a trigger (similar to the *memory access* mechanism's trigger) and silently load data within the dropped network packet as a malicious firmware that runs within shadow mode.

Figure 1 shows the difference between running in normal mode before a shadow firmware has been loaded, and running in shadow mode. In normal mode, the processor goes through the cache hierarchy. While running in shadow mode, the processor limits what activity will go out to the memory bus. A portion of the address space is backed only by the L1 cache. Instruction fetches are satisfied from a small reserved portion of the icache, and data loads/stores are satisfied from a reserved portion of the dcache. If the malicious service needs to access regular memory then it issues loads or stores outside of the reserved address space. Otherwise, if it keeps within the reserved space, then it won't generate any activity on the memory bus, and is almost entirely invisible. Not pictured is normal execution while a malicious firmware is loaded; this is the same as normal execution, except that a portion of the icache and dcache are unavailable (as they are holding instructions and data for the malicious program).

When the attack is running, the shadow-mode code must be able to gain control of the processor at key points in the instruction stream. To support transitions into shadow mode, we use the debugging hardware found on many processors, which includes breakpoints and watchpoints typically. We extend typical watchpoints to allow shadow-mode code to trap on data values in addition to addresses since many of our services use this trigger as part of the attack.



Attacker sends unsolicited UDP packet

Monitor notices the magic byte sequence

Firmware is copied to reserved cache area and activated.

```
 login: root
 password: letmein

 Last login: Mon Apr 1
[root@victim ~]$
```

Attacker logs in as root. Shadow firmware uninstalls.

Figure 2: Overview of the login attack.

Using shadow mode, we implement two attacks; a backdoor service (Section 4.2) and a password sniffer (Section 4.3).

4 Example malicious services

In this section we discuss the malicious services we implement and the tradeoffs we make in our designs. We discuss a malicious service that escalates the privileges of a process, a malicious service that allows attacker to automatically login to a system, and a malicious service that steals passwords.

4.1 Privilege escalation

Using the *memory access* mechanism, we implement a malicious service that escalates the privileges of a user process to root privilege level. To carry out the attack, our privilege escalation program uses our trojaned hardware to turn off protection to privileged memory regions. Then, it searches kernel memory looking for its own process structure, and it changes its effective user ID to root so it runs with full system privileges.

This attack uses simple hardware mechanisms to allow malicious software to gain control of the system without exploiting a bug in software. Our memory access mechanism increases our logic-gate count by only 0.05%, yet it allows us to violate directly OS assumptions about memory protection, giving us a powerful attack vector into the system.

4.2 Login backdoor

Using the *shadow mode* mechanism, we implement a malicious service that acts as a permanent backdoor into a system (Figure 2). To initiate the attack, an attacker sends an unsolicited network packet to the target system and the target OS inspects the packet to verify the UDP checksum. The act of inspecting the packet (necessary to decide if it should be dropped) triggers the trojaned hardware, and the malicious service interprets the contents of the packet as new firmware that it loads into the processor invisibly. The target operating system then drops the unsolicited packet and continues operation, oblivious to the attack.

The shadow mode firmware monitors the `login` application. When it detects a user trying to login with the password "letmein", the malicious service modifies the return value of the password checking function to return true, granting access to any user who uses this password. To reduce the footprint of the attack, after a successful login attempt the firmware unloads itself and turns off shadow mode, returning all processor resources to the system. By sending then UDP packet and then immediately attempting to login, an attacker requires shadow mode to be active for a minimal time span.

This network-based technique for injecting attacks has two key advantages. First, this technique is flexible since the attack itself can be updated via the network. This flexibility allows attackers to adjust the attack to cope to a changing software environment, or to install a completely separate attack. Second, this technique avoids adding extra states and events to the system that are visible from outside of the CPU. We detect the sequence of bytes during the UDP checksum calculation, so the attack data has a high probability of already being present within the data cache. Then, the bootstrap process installs this data within reserved cache lines, so no off-chip memory accesses are needed to initialize the attack, even after the OS drops the packet. For systems without network access, similar external data can be used (e.g., the first block of a USB key, needed to identify the filesystem type).

The net effect of this attack is that an attacker can send one attack packet to a system to enable the "letmein" password, and then login to any account (including root). The attacker can then use any traditional methods of manipulating the system to avoid detection and to carry out malicious activities. The shadow mode mechanism increases the logic-gate count by only 0.08%, and gives us unlimited access to the machine without exploiting a software vulnerability.

4.3 Stealing passwords

Again using the *shadow mode* mechanism, we implement a service that steals passwords from users on the system. Since the processor can ignore software memory protection at will, the primary difficulty is finding the passwords within a swamp of random data. Our service first interposes on the `write` library call, searching for the string "Password:" to identify processes that are likely to receive passwords. On the following `read` call it interposes to record potential passwords.

To find the `read` and `write` functions and interpose on their invocations, we exploit the fixed interface of the ELF binary file format. A process that is linked against shared libraries will have sections of its executable file devoted to the list of library functions it needs. Conveniently, one of those sections includes a string table, which contains the human meaning — the names — of the library functions. Whenever the IMP encounters a new process, it switches control to the service, which parses the executable section of the new process looking for the parts of the ELF file needed by the dynamic linker. With this information it can determine first the virtual and then the physical addresses of the library functions to interpose on. By setting breakpoints at the physical addresses, it can find shared library calls regardless of the virtual address of the caller.

Once we steal passwords, we use two different techniques to leak passwords out of the system via the network. Our first technique uses system calls to access the network interface. This technique for accessing the network is hardware independent, and relies only on the (relatively static) system call interface. The disadvantage of this technique is that it requires interactions with the operating system that will result in visible attack states and events. We could reverse state changes by using an undo log [7], but restoring system-level checkpoints would cause timing perturbations. Instead we make some effort to clean up (e.g., close sockets), but OS-level memory perturbations resulting from our technique remain.

A second technique we implement is to overwrite existing network frames with our own packets. These packets are UDP packets with a predetermined destination IP address and data that contains the stolen passwords. To overwrite packets with our malicious UDP packets, we must first identify events that indicate the driver is sending a new transmit packet to the device. To identify new transmit packets, we interpose on memory-mapped I/O between the driver and the device control registers. For our network card, the driver writes a command to the card telling it to allocate a new transmit buffer. Then, software reads the return value from this allocate. At this point we know any subsequent writes to the allocated buffer will be for a new network packet. To overwrite the driver's packet, we interpose on writes to the data control register. Our network card uses memory-mapped I/O to transfer data so we can modify each store instruction directly. If the network card had supported direct memory access (DMA), we could have stalled the write to the control register that signals the start of a DMA transfer. While this write is stalled, we could modify the physical memory used for DMA. Of course we would have to take special care to prevent software from reading our modified memory, and we would have to restore the original data after the transmit completes.

Since this attack must have shadow mode continuously active, it will impose higher timing perturbations than the previous attacks. We quantify the time overhead of persistent shadow-mode attacks in Section 6.

5 Implementation

This section describes the implementation of our IMPs and the three malicious services we used to evaluate our malicious processors.

We implement our malicious processors on an FPGA development board using a modified version of the Leon3 processor [8]. The Leon3 processor is an open source SPARC processor that implements the SPARC v8 instruction set and includes a memory management unit (MMU). We modify the design of the processor at the VHDL level, and synthesize and run it on the FPGA. Our system runs Linux and has many of the peripherals one would find on a typical computer system, such as Ethernet, USB, VGA, and PS/2.

To support attacks we implement the full memory access mechanism, and most of the shadow mode mechanism using our FPGA-based system. Our shadow mode implementation supports the loading and running of invisible malicious firmware. Also, we modify the watchpoint hardware to trap on values instead of addresses, with the watchpoint exception routing code using a small bootstrap software stub (as described in Section 3.2).

We evaluate our privilege escalation attack (Section 4.1) and our service for automatic login (Section 4.2) on our FPGA-based system. Also, we us a full-system functional simulator for more detailed analysis of the automatic login attack, and to evaluate the password stealing attack (Section 4.3).

6 Evaluation

This section evaluates the impact of an IMP on a system. Using our hardware-based testbed, we evaluate the circuit-level impact of our designs and we measure the runtime performance impact of our shadow mode mechanism. We measure the performance impact of shadow mode since it is the only mechanism that perturbs the timing of the system.

6.1 Circuit-level perturbations

Processor	Logic gates	Lines of VHDL code
baseline CPU	1,787,958	11,195
CPU + memory access	1,788,917	11,263
CPU + shadow mode	1,789,299	11,312

Table 1: This table summarizes the circuit-level impact of our IMPs compared to a baseline (unmodified) Leon3 processor. We show the impact of an IMP that includes our memory access mechanism and an IMP that includes our shadow mode mechanism.

Table 1 summarizes the circuit-level impact of implementing our memory access mechanism and implementing our shadow mode mechanism using our modified Leon3 processor. To implement our memory access mechanism, we modify the data cache and the MMU so that memory permission checks are ignored for malicious software. To implement our shadow mode mechanism we modify our instruction and data caches to reserve one set in each cache for attack code and attack data, after an attack has been initialized. Also, we add a new type of watchpoint that traps on load/store values instead of addresses, and we make some minor changes to the existing watchpoints to make them suitable for use in shadow mode. We synthesize our designs using the Xilinx ISE 9.1i tool chain, which is used to create images that run on our FPGA. Our memory access mechanism adds 959 logic gates to the baseline processor and our shadow mode modifications add 1341 logic gates. This 0.05% and 0.08% increase in logic is likely to decrease for larger processor that include up to one billion logic gates. Also, we make few changes to the VHDL code of the processor; we add 68 lines of code for our memory access mechanism and 117 lines of code for our shadow mode mechanism.

Our mechanisms do not affect the timing of our processor. Using the Xilinx tool chain, we perform a detailed timing analysis of our circuits and we verify that our designs are off the critical path. Our IMPs run at 40 MHz, which is the recommended clock speed for our hardware configuration.

6.2 Timing perturbations

To test timing perturbations we use our FPGA-based system. Our processor runs at 40 MHz, has a 2-way set associative 4k instruction cache, a 4-way set associative 8k data cache, and 64 MB of main memory, which is comparable to a mid-level embedded system.

Figure 3 shows the timing effects of various attack conditions. We run various benchmarks; four CPU bound SPEC benchmarks bzip2, gcc, parser, and twolf, as well as an I/O bound benchmark wget, where we repeatedly downloaded files from a remote system. As the embedded system was rather small (64MB of RAM) and runs a fully-featured operating system (Linux), this represents every SPEC CINT2000 benchmark that had suf-

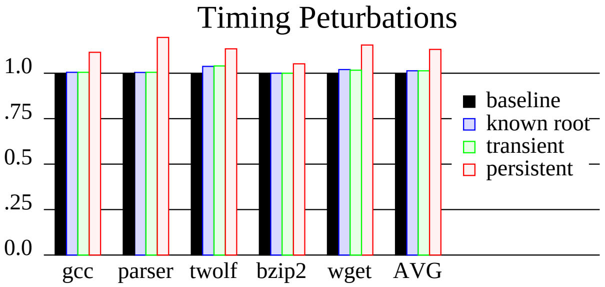


Figure 3: Time perturbations are measured relative to the baseline (non-attack) tests.

ficiently small inputs to successfully run. We had four experimental cases, respectively:

Baseline: Run the benchmarks on unmodified hardware and without attacking.

Known Root: Run the benchmarks on unmodified hardware, but attack the system in the middle. We use the proper root password to log in and steal the /etc/shadow file. This represents the best scenario an attacker could hope to have; they already have proper credentials, and all they need to do is deliver their payload. This imposes 1.32% overhead over the base case (arithmatic average).

Transient: Run the benchmarks on hardware with shadow mode support. In the middle of the benchmarks, we used the hardware login backdoor (described in Section 4.2) to login and steal the /etc/shadow file. This imposes an average of 1.34% overhead compared to the baseline; this is merely 0.0235% over the known root case. The underlying payload is by far the most expensive portion of the attack. This "hit-and-run" style attack minimized the amount of time that shadow mode is active since the attack cleans itself up after successful logins. Hence, all hardware resources are available for non-malicious jobs and the overhead is minimized.

Persistent: Run the benchmarks on hardware with shadow mode support, and have the hardware login backdoor continuously active. Shadow mode is active during the entire run; it is never disabled. This imposes an average of 13.0% overhead compared to the baseline, or 11.5% compared to the known root attack.

7 Defending against malicious processors

Unsurprisingly, current defense techniques are completely ineffective against malicious processors. Fortunately, defense is mainly a matter of detection, and malicious processors differ from their benevolent counterparts in several ways. This section covers detecting malicious processors via the analog and digital perturbations they introduce. Since the malicious processor designer can tradeoff among these, the best defense is most likely a combination approach.

7.1 Analog side effects

The possibility of using power analysis to identify malicious circuits was considered in recent work by Agrawal, *et al.* [3]. However, power analysis began as an attack technique [12]. Hence there is a large body of work to preventing power analysis, especially using dual-rail and constant power draw circuits [17, 18]. For the designer of a trojan circuit, such countermeasures are especially feasible; the area overheads only apply to the small trojan circuit, and not to the entire processor.

Analog side effects can also be used to identify individual chips. Process variations cause each chip to behave slightly different, and Gassend, *et al.* use this fact to create physically random functions (PUFs) that can be used to identify individual chips uniquely [9]. Chip identification ensures that chips are not swapped in transit, but chip identification can be avoided by inserting the malicious circuits higher up in the supply pipeline.

7.2 Digital perturbations

Current testing, reverse engineering, and fault-tolerance techniques are unsuitable for detecting digital side effects resulting from compromised ICs. IC testing [19] enumerates through various input states and transitions, and measures the output to confirm that the IC is performing correctly. The skillful malicious processor architect has several weapons to try to avoid digital side effects. Waiting for a period of time is the most straightforward, although this could cause problems depending on how long the tests run. Waiting for a specific input will almost certainly pass testing, but now the attacker must induce this sequence on deployed systems. Given the large state space of a microprocessor[1], it is quite possible that digital testing will fail to discover malicious circuits even if the attacker takes no precautions.

IC reverse engineering can re-create complete circuit diagrams of manufactured ICs. However, these techniques are time consuming, expensive, and destructive, taking up to a week to reverse engineer a single chip and costing up to $250,000 [13]. Thus, these techniques can be used on only a small percentage of samples, providing malicious ICs an opportunity to escape such techniques.

Fault-tolerance techniques [14, 6] may be effective against malicious ICs. In the Byzantine Generals problem [14], Lamport, *et al.* prove that 3m+1 ICs are be needed to cope with m malicious ICs. Although this technique can be applied in theory, in practice this amount of redundancy may be too expensive because of

[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.

cost, power consumption, and board real estate. Furthermore, only 59 foundries worldwide can process state-of-the-art 300mm wafers [16], so one must choose manufacturing locations carefully to achieve the diversity needed to cope with malicious ICs.

8 Conclusions

In this paper we have laid the groundwork for constructing malicious processors capable of valuable, high level, sophisticated attacks. We argue that the IC supply chain is large and vulnerable, and that there are organizations with the competence, resources, and motivation to build and deploy malicious circuits. We implemented two general purpose mechanisms for designing malicious processors, and used them to implement attacks that steal passwords, enable privilege escalation, and allow automatic logins into compromised systems.

We showed that portions of this design space are surprisingly low in the amount of hardware and the possibility of detection, while still allowing impressive attacks. The login attack used only 1341 additional gates, yet gave an attacker complete and high-level access to the machine. This same hardware could support a wide variety of attacks and is flexible enough to support dynamic upgrades.

Overall, we found that malicious processors are more practical, more flexible, and harder to detect than an initial analysis would suggest; malicious hardware deserves its share of research attention.

Acknowledgment

We would like to thank Frans Kaashoek and Milo Martin for suggesting we look at privilege escalation. We would like to thank Jonathan Smith for pointing us to the DoD report on high performance IC supply, and we would also like to thank Vikram Adve, Landon Cox, Michael LeMay, and Ed Nightingale for feedback on an early draft of this paper. Finally, we would like to thank the anonymous reviewers for their helpful feedback.

References

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

[2] The IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide. Technical report, Intel Corporation, 2004.

[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] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault-Tolerance. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 1–11, December 1995.

[7] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3), September 2002.

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

[9] 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.

[10] D. E. Hoffman. CIA slipped bugs to Soviets. *The Washington Post*, February 2004. http://www.msnbc.msn.com/id/4394002.

[11] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 314–327, May 2006.

[12] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397, London, UK, 1999. Springer-Verlag.

[13] J. Kumagai. Chip detectives. *IEEE Spectr.*, 37(11):43–49, 2000.

[14] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[15] P. A. Layman, S. Chaudhry, J. G. Norman, and J. R. Thomson. United States Patent 6906962: Method for defining the initial state of static random access memory.

[16] U. S. D. of Defense. Defense science board task force on high performance microchip supply. February 2005. http://www.acq.osd.mil/dsb/reports/2005-02-HPMS_Report_Final.pdf.

[17] D. Sokolov, J. Murphy, A. Bystrov, and A. Yakovlev. Design and analysis of dual-rail circuits for security applications. *IEEE Trans. Comput.*, 54(4):449–460, 2005.

[18] K. Tiri and I. Verbauwhede. Design method for constant power consumption of differential logic circuits. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 628–633, Washington, DC, USA, 2005. IEEE Computer Society.

[19] N. H. Weste and K. Eshraghian. *Principles Of CMOS VLSI Design, A Systems Perspective*. Addison Wesley, 1993.