OS Support for Detecting Trojan Circuit Attacks

Gedare Bloom, Bhagirath Narahari, and Rahul Simha
Dept. of Computer Science,George Washington University, Washington, DC 20052
E-mail: {gedare,narahari,simha}@gwu.edu, Tel: (202) 994–7181, Fax: (202) 994–4875

***Abstract*—Rapid advances in integrated circuit (IC) development predicted by Moore's Law lead to increasingly complex, hard to verify IC designs. Design insiders or adversaries employed at untrusted locations can insert malicious Trojan circuits capable of launching attacks in hardware or supporting software-based attacks. In this paper, we provide a method for detecting Trojan circuit denial-of-service attacks using a simple, verifiable hardware guard external to the complex CPU. The operating system produces liveness checks, embedded in the software clock, to which the guard can respond. We also present a novel method for the OS to detect a hardware-software (HW/SW) Trojan privilege escalation attack by using OS-generated checks to test if the CPU hardware is enforcing memory protection (MP). Our implementation of fine-grained periodic checking of MP enforcement incurs only 2.2% overhead using SPECint 2006.**

I. Introduction

The economic strain of fabricating at ever smaller scales has led to stratification of the semiconductor industry. IC design, fabrication, verification, and integration are often conducted at different sites under different authority which increases risk by requiring multi-party trust. Because of these risks, malicious parties can threaten the security of the IC design and fabrication processes (the IC supply-chain).

One security threat to the IC supply-chain is the *Trojan circuit* (also known as hidden malicious circuit and hardware Trojan), a malicious insertion to or modification of an IC that can occur any time between design and fabrication. An attacker can use a Trojan circuit to carry out attacks which, if undetected, may easily compromise the device relying on the subverted IC. Initial research in to Trojan circuits focused primarily on hardware-only attacks. Two particular Trojan-enabled attacks, denial-of-service and (sensitive) information leakage, dominate the literature. DoS attacks appear to have garnered the most attention [1]; however, King et al. [2] demonstrate more intricate attacks that involve malicious software interacting with the Trojan circuit.

We use the term *HW/SW Trojan* for situations involving the collusion of malicious software with Trojan-infected hardware, or where attackers as King et al. put it, "design hardware to support attacks" [2] rather than hard-code the attack itself in hardware. Two hardware mechanisms for HW/SW Trojans were presented by King et al., a memory access mechanism and a shadow mode mechanism. Leveraging the memory access mechanism, they created malicious software that escalates its privilege to that of the superuser (root). Two software services were implemented using the shadow mode mechanism, one to insert a login backdoor and the other to steal passwords. To defend against these attacks, King et al. suggest detection by observing the analog and digital perturbations introduced by Trojan circuits.

The main focus of our work is on defending against such HW/SW Trojans; specifically, we target the memory access mechanism with the privilege escalation attack. A secondary contribution is in detection of restricted denial-of-service (DoS) attacks. We assume a relatively simple Trojan circuit that does not authenticate the malicious software. On one hand, if the Trojan circuit knows when malicious software is running, then attacks could be much harder to detect. On the other hand, such a Trojan circuit would be more complex, and thus larger and arguably easier to detect using side-channel analysis.

Current defense techniques for Trojan circuits primarily involve IC verification through testing and side-channel analysis. We propose to add another complementary layer of techniques to provide runtime detection for those Trojan circuits that slip past the chip verification and test phase. In particular, we assume a trusted operating system which we modify to create two types of checks on the hardware. First, we create liveness checks in the form of randomized off-chip accesses to a simple hardware module, called a *guard*. We assume the guard hardware is verifiable due to its simplicity, and we use this simpler, off-chip hardware to monitor behavior of a very complex IC. Second, we add memory protection (MP) checks as code executing in unprivileged mode that attempts to access privileged memory areas. Liveness checks provide restricted DoS detection, and MP checks detect the privilege escalation attack discussed above.

Checks produced by the OS have an impact on the system's runtime performance. Liveness checks introduce minimal overhead because they occur as part of the existing periodic interrupt used by commodity OSs to manage task switching. However, executing checks may introduce a lot of overhead depending on the mechanism and periodicity of checking MP mechanisms – one of the goals in our research is to evaluate and to minimize such overhead. In order to determine how often MP enforcement must be verified, we re-implemented the privilege escalation attack and found that malicious software only needs MP mechanisms to be turned off for about 15–20 μs on our system. To provide such fine-grained checking, we use a real-time OS extension to Linux and we achieved 2.2% overhead averaged across SPECint 2006. These results are shown in Section III. We next present the details of the OS-generated liveness and MP checks.

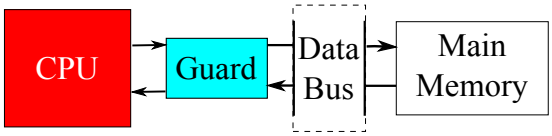

Fig. 1. System Architecture. A verifiable hardware module (guard) is added to the memory hierarchy. The OS will use the guard to verify that the CPU is trustworthy.

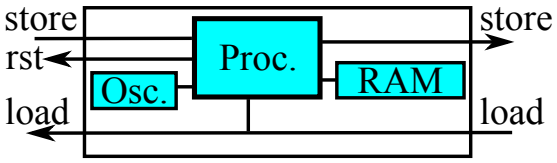

Fig. 2. Hardware Guard Internals. The guard contains an oscillator, storage space, and processing logic. If an attack is detected, the guard can reset the CPU (via the RST line) or can provide notification to an external entity.

II. OS Checks for Hardware Verification

At a high level, our solution is conceptually simple. As shown in Fig. 1, we add an off-chip *guard*, which is made of easily verifiable hardware, to implement light-weight checks of the untrusted CPU. For detecting DoS, we introduce liveness checks that go off-chip to the guard at pseudorandom intervals by using the OS timer interrupt. To detect the privilege escalation attack, we adopt real-time OS (RTOS) mechanisms for MP checks. By modifying the OS to be guard-aware, the OS can check for HW/SW Trojan attacks.

A. Assumptions and Scope

We make a few key assumptions in our research. First, the OS and guard are presumed to be trusted and verified. Second, the guard is assumed to have a precise notion of timing, using an oscillator as shown in Fig. 2. Third, non-cacheable accesses to memory can be made by the OS, which is possible with most cache controllers and is exposed by most OSs, for example the ZONE_DMA area of Linux.

Some limitations to the scope of our research are also worth noting. First, we only attempt to protect against two particular attacks, DoS and privilege escalation (through disabled MP). Further, the DoS detection is restricted to verifying that the OS is receiving timer interrupts. Second, we focus only on the CPU as being possibly malicious – peripherals are trusted.

B. Hardware Guard

The guard is a verifiable hardware module that is placed off-chip and provides a verifier for the OS to test the CPU. Fig. 2 shows the internal components of the guard, including an oscillator for timing, scratch RAM for storing pseudorandom values, and a simple processor. The guard observes memory accesses to detect OS-generated checks. A timer is managed by the processing logic and synchronized with the CPU's timer interrupts. When a check is received, a watchdog (countdown) timer is set to a pseudorandom value. If the watchdog times out, the guard detects an attack and can reset the CPU or notify another entity (e.g. a human).

C. Liveness Checks

For DoS detection we add liveness checks, a similar concept to the heartbeats from our prior work [3], but now implemented in the OS instead of as part of applications. These liveness checks are pseudorandom non-cached memory accesses, which prevents simple prediction, delay, and replay attacks. A significantly delayed liveness check triggers DoS detection. Note that by liveness we mean that the CPU is providing the OS with correct timing interrupts; we can check for liveness because the guard can synchronize with the expected time interrupts of the CPU. This notion of liveness is overly simplistic (and ignores progress), and thus the DoS detection is restricted.

The OS generates the liveness checks as part of the regular timer-interrupt. In our implementation, the liveness checks were added to the `do_timer` function in Linux, which is invoked every 1 ms to update the software clock and perform various kernel bookkeeping tasks. `do_timer` is architecture-independent, so our modification is fully portable and noninvasive. Before the number of expected ticks have elapsed since the last liveness check was sent, the OS sends a fresh check. The overhead of liveness checking is minimal when aligned with the existing periodic interrupt used by commodity OSs to manage task switching.

As each liveness check is received by the guard, the guard determines how many ms until the next check should arrive by obtaining the next value in the pseudorandom sequence. Computing this sequence is not time-critical, because at least 1 ms will elapse before the next liveness check might be expected. Thus computing the next value in the sequence can be complex without degrading performance or security.

An obvious problem is in providing the OS with the ability to generate the sequence without exposing the pseudorandom seed (or state) to the Trojan circuit. This problem is actually an instance of a more general problem: How to hide a variable from a Trojan circuit. We do not yet have a complete solution to this problem. By randomizing the location of the variable it can be hidden from simple Trojan circuits, but loading the variable to a register may expose the value. Another possibility is for the guard to instrument binary re-writing of the OS code that implements the liveness check, thus allowing the values to be inserted indirectly in the control flow. The problem of securing a variable from a Trojan circuit is an area that requires further research. We suspect obfuscation techniques might increase the burden so that any Trojan circuit capable of reverse-engineering the obfuscation will be detectable by side-channel analysis.

D. Memory Protection (MP) Checks

King et al.' privilege escalation attack is a straightforward HW/SW Trojan attack. First, the Trojan circuit is somehow triggered by software to turn off memory protection (MP); without MP, the memory space of the OS can be accessed by any process. Next, malicious software accesses the OS's process list, shown in Fig. 3 and searches for its own process control block (PCB).

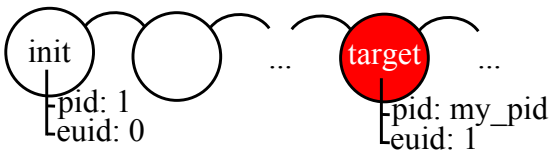

Fig. 3. Privilege Escalation Attack. Malicious software walks the list searching for its own process control block (PCB), then sets the effective user ID (EUID) to the superuser, which is 0 for Linux.

By changing the effective user ID (EUID) field of the PCB to be the superuser, the SW Trojan elevates its privilege; turning MP back on closes the vulnerability and makes detection harder. The following C-like code demonstrates the attack:

```
task = 0xc07093e0;
pid_offset = 464;
euid_offset = 676;

/* tell Trojan circuit to disable MP */
disable_prot();

/* find the PCB with my_pid */
do {
  task = next_task(task);
} while (*(task+pid_offset) != my_pid);

/* privilege escalation */
*(task+euid_offset) = 0;
enable_prot();
```

Here `task` is initialized to the head of the process list; in Linux, this is the init_task symbol in `/proc/kallsyms`. Values for `pid_offset` and `euid_offset` are the offsets in the PCB of the process ID and the EUID. `disable_prot` is how the SW Trojan requests the Trojan circuit to disable MP. The SW Trojan searches the process list until finding the PCB with a process ID matching `my_pid`, sets the EUID to be 0 (root on Linux), then requests MP be enabled.

We implemented the above code, and searching the list takes about 15–20μs under low load. Thus, one of our goals is to check at least every 15μs that MP protection is still on. Because Linux provides at most 1000 Hz frequency for scheduling, a full millisecond elapses between scheduling events. Therefore, we chose to investigate real-time scheduling in Linux as a way to achieve fine-grained verification.

We use a RT scheduler that can schedule tasks with a fixed period and duration. We chose the Xenomai Real-Time Framework for Linux [4], which provides RT scheduling. Fig. 4 shows how Xenomai fits in to the hardware-software interface. Xenomai is implemented as a patch of Linux plus a library to access the RT infrastructure. If there are no RT tasks to schedule, Xenomai invokes the Linux scheduler. RT tasks have limited library support, restricted to code exported by Xenomai and a limited subset of the Linux kernel API. This limitation does not affect our solution, because the MP check only requires a raw memory accesses.

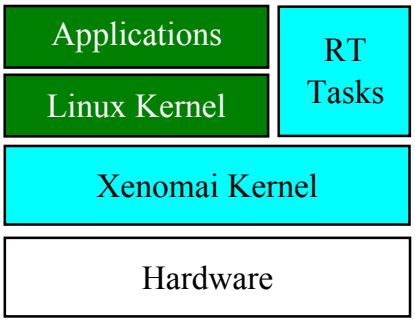

Fig. 4. Xenomai Extension to Linux. Xenomai conceptually layers a thin kernel under Linux that can provide real-time (RT) scheduling of RT tasks. In reality, Xenomai extends the Linux kernel and is integrated as part of the OS, which reduces potential performance overheads of switching in and out of the Xenomai kernel functionality.

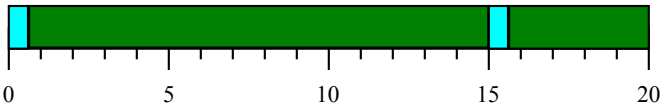

Fig. 5. RT Task Scheduling in Xenomai (μs scale). The MP check is a RT task with a period of 15μs, shown in light blue. When the RT task is not being executed, Xenomai invokes the regular Linux scheduler which executes tasks as normal, shown in dark green.

We created a RT task using `rt_task_set_periodic` with period 15μs. The RT task attempts to read memory in the kernel address space – an MP check – and then sleeps by calling `rt_task_wait_period`. Fig. 5 shows the runtime behavior of scheduling this task (in light blue) and for the rest of the software stack, including Linux and applications (in dark green).

III. Evaluation

Liveness checks in the regular OS clock provide detection of DoS with a few instructions in the timer interrupt. Memory protection (MP) checks, in the form of user-space attempts to access directly the kernel's address space, provide detection of a privilege escalation attack window. We measure the overhead of our solution on commodity hardware with SPECint 2006 [5].

We evaluated the performance of our system using Linux and Xenomai. All experiments were conducted on an Intel Core-2 2.0 GHz with 2 GB RAM, running Linux version 2.6.25.11 in Fedora Core 7. We removed all non-essential modules from the Linux image, and we disabled power management functions. This was the baseline against which our results are normalized. For the liveness checks, we added instructions to the software timer interrupt handler `do_timer`. For the RT extensions to Linux, we used Xenomai version 2.4.4 and we ran MP checking tasks pinned to each core.

We ran all of the SPECint benchmarks three full runs (reportable) and took the median of the three runs per benchmark. We also give an average across these medians. The SPEC benchmarks were compiled with `-O2`.

Fig. 6 shows the overhead for the Xenomai extension (RTOS) both with and without issuing MP checks. The results

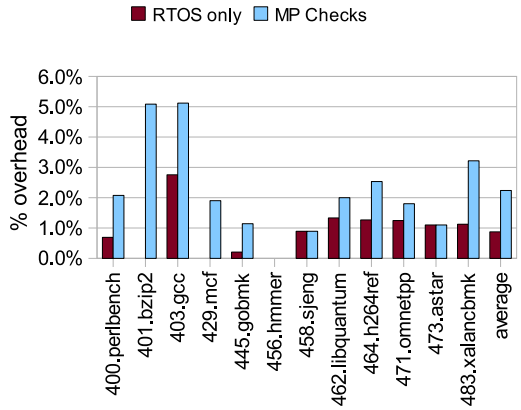

Fig. 6. Overhead of Real-Time OS Solution Compared to Unmodified Linux. By adding real-time scheduling capabilities to the OS, low-overhead software checks can be periodically scheduled for every 15–20μs. MP Checks shows that using RTOS support for enforcement checking incurs 2.2% average overhead compared to unmodified Linux.

show that Xenomai alone added 0.9% average overhead to the unmodified kernel, and that Xenomai with the MP checks adds 2.2% overhead on average. The largest overhead for MP checks was 5.1% and the lowest was negligibly small.

Although we have focused on the privilege escalation attack and MP checks, extending our solution to other privileged accesses is trivial. We implemented checks executing privileged instructions without having the highest current privilege level (CPL). These CPL checks are similar to MP checks: Short enough to have low overhead and verifiable by a hardware guard that is monitoring the instruction stream.

IV. Related Work

Two primary methods of detection are current research trends in Trojan detection: logic-based testing [6]–[9] and side-channel analysis [10]–[13]. Other literature in the field discusses alternate defense methods [3], [14], [15]

Our current and prior work [3], in contrast to most other techniques, focuses on detecting Trojan circuits in deployed devices. In other words, we provide another layer of defense in case a Trojan circuit avoids detection. On-line detection is by no means our contribution: others have discussed measuring and reporting physical characteristics with on-chip sensors [14]. Similarly, the time-tested replication of entire processing elements can help to detect some Trojan circuit attacks [2], but the overhead is likely to be unbearable for full state-machine replication. Our approach is complementary to existing runtime techniques, and our research focuses on minimizing the overhead of runtime checks for Trojan circuits.

V. Conclusion

The area of runtime detection of Trojan circuits is rife with research opportunities. By adding liveness checks to Linux, we are able to provide detection of some Trojan circuit DoS attacks. We have also shown how to detect a privilege escalation HW/SW Trojan attack through the use of periodic checks of memory protection mechanisms using a RT extension to Linux. Using a RTOS for generating checks provides an interesting platform for the OS to detect HW/SW Trojan attacks. By timing how large of an attack window a particular exploit requires, the OS can probe the hardware to check for runtime vulnerabilities. Although our solution is just a patch for a known attacks, the defense technique we propose is effective, efficient, and novel. Our results are encouraging, with a low 2.2% average overhead for periodic checking of the memory protection mechanisms and negligible overhead for the DoS detection. These detection techniques are compatible with existing Trojan circuit detection techniques.

Acknowledgment

This work is partially supported by NSF grants ITR-025207 and CNS-0934725, and AFOSR grant FA955006-1-0152.

References

[1] S. Adee, "The hunt for the kill switch," *Spectrum, IEEE*, vol. 45, no. 5, pp. 34–39, 2008.

[2] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware," in *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*. San Francisco, California: USENIX Association, 2008, pp. 1–8.

[3] G. Bloom, B. Narahari, R. Simha, and J. Zambreno, "Providing secure execution environments with a last line of defense against trojan circuit attacks," *Computers & Security*, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.cose.2009.03.002

[4] P. Gerum, "Xenomai - implementing a RTOS emulation framework on GNU/Linux," 2004. [Online]. Available: http://www.xenomai.org/documentation/branches/v2.3.x/pdf/xenomai.pdf

[5] Standard Performance Evaluation Corporation, "SPEC CPU2006." [Online]. Available: http://www.spec.org/cpu2006/

[6] S. Smith and J. Di, "Detecting malicious logic through structural checking," in *Region 5 Technical Conference, 2007 IEEE*, 2007, pp. 217–222.

[7] F. Wolff, C. Papachristou, S. Bhunia, and R. S. Chakraborty, "Towards trojan-free trusted ICs: problem analysis and detection scheme," in *Proceedings of the conference on Design, automation and test in Europe*. Munich, Germany: ACM, 2008, pp. 1362–1365.

[8] R. Chakraborty, S. Paul, and S. Bhunia, "On-demand transparency for improving hardware trojan detectability," in *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, 2008, pp. 48–50.

[9] S. Dutt and L. Li, "Trust-Based design and check of FPGA circuits using Two-Level randomized ECC structures," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 1, pp. 1–36, 2009.

[10] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan detection using IC fingerprinting," in *Security and Privacy, 2007. SP '07. IEEE Symposium on*, 2007, pp. 296–310.

[11] R. Rad, J. Plusquellic, and M. Tehranipoor, "Sensitivity analysis to hardware trojans using power supply transient signals," in *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, 2008, pp. 3–7.

[12] J. Li and J. Lach, "At-speed delay characterization for IC authentication and trojan horse detection," in *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, 2008, pp. 8–14.

[13] Y. Jin and Y. Makris, "Hardware trojan detection using path delay fingerprint," in *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, 2008, pp. 51–57.

[14] X. Wang, M. Tehranipoor, and J. Plusquellic, "Detecting malicious inclusions in secure hardware: Challenges and solutions," in *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, 2008, pp. 15–19.

[15] M. Banga and M. Hsiao, "A region based approach for the identification of hardware trojans," in *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, 2008, pp. 40–47.