# Privilege escalation attack through address space identifier corruption in untrusted modern processors

Michail Maniatakos
Electrical and Computer Engineering Department
New York University Abu Dhabi
Abu Dhabi, UAE
michail.maniatakos@nyu.edu

***Abstract*—Privilege escalation attacks are one of the major threats jeopardizing microprocessor operation. Such attacks aim to maliciously increase the privilege level of the executed process, in order to access unauthorized resources. Modern microprocessors include complex memory management modules, with various different privilege levels and numerous ways to change the privilege level. In this paper, we present a malicious modification in the microprocessor process switch mechanism. Contrary to recent work presented in literature, the modification can be deployed during manufacturing process, as it consists of a trivial addition of a gate or wire sizing. The minimal footprint, however, comes at the cost of small window of attack opportunities. Experimental results show that a modification-aware application can gain escalated privileges within a few thousand clock cycles. Moreover, the malicious code has been added to SPEC benchmarks, and we show that the modified benchmarks can get escalated privileges before the end of typical workload, with minimal performance overhead.**

## I. Introduction

Modern microprocessors are ubiquitously deployed in a wide range of applications: From personal computers, laptops and cellphones, to space and automotive applications. Therefore, ensuring the integrity of a microprocessor is of critical importance, as security breaches can have catastrophic impact. Microprocessor designers incorporate extended security features in latest designs, in an effort to protect the system from external attacks. Due to the globalized supply chain, however, the final design may be tampered with and not satisfy the security properties specified by the designers.

Due to the vast choice of intellectual property (IP) cores, circuit designers and system integrators are highly liberated from the tedious work to design functional modules but only to focus on the development of system architectures. However, the widely usage of IP cores also breeds new security problems. Before arriving at the hand of the system integrator, an IP core has traveled through many stages and is modified by various design houses [1]. There are plenty of opportunities for attackers to insert malicious logic in the IP core throughout the whole IP transaction process. Such modifications, known as hardware Trojans, are purportedly done without the knowledge of the IP consumer. The additional functionality can be exploited by a perpetrator to cause catastrophic results if the IP core is embedded into mission-critical device.

Especially in the domain of modern microprocessors, as discussed previously, malicious modifications can have extensive impact: The dramatic increase of mobile devices, consisting of high-performance, multi-core microprocessors, provide a wide radius of attack impact. Due to the microprocessor complexity, however, introducing malicious modifications is a very tedious process, as microprocessor designs are very compact with very limited, if any, area for malicious hardware. Therefore, it is very difficult to maliciously insert a significant amount of hardware at the design stage of the microprocessor.

In this paper, we present a malicious modification with negligible overhead. Contrary to recent work discussed in Section II, this modification can be applied during the fabrication stage of the microprocessor, as it can be as simple as an addition of a buffer or changing the size of a wire. In order to stage the attack, a malfunction is introduced on the memory subsystem of the microprocessor, allowing a compromised process to access unauthorized memory locations and gain escalated privileges. The attack is extensively discussed in Section III. The rest of the paper is organized as follows: Section IV presents the experimental platform used to demonstrate the effectiveness of the attack. As due to the negligible overhead, there is limited control over the attack, experimental results presented in Section V corroborate that compromised processes can gain escalated privileges before the end of their execution. Finally, conclusions and future directions of the presented work are discussed in Section VI.

## II. Related work

Malicious modification insertion and detection has risen as a contemporary topic of interest. An extensive taxonomy of modifications (called "Hardware Trojans" appears) in [2]. Side channel attacks, such as wireless channels, have also been extensively discussed [3], [4], [5] for integrated circuits (ICs). Nonetheless, there has been a limited amount of work targeting commercial, high performance microprocessors.

A recent competition (Cyber Security Awareness Week - CSAW [6]) targeted an i8051 microprocessor executing a cryptographic algorithm. The winners presented a set of malicious modifications that allow the attacker to execute unauthorized code [7]. As the competition targeted an old microprocessor and no operating system was used, the solutions can not be easily ported to latest commercial microprocessors.

The authors of [8], present a microprocessor modification targeting advanced systems. The authors insert extra hard-

978-1-4673-6040-1/13/$31.00 

ware during the design stage, providing the attacker with an extended stage of attack (login backdoor, stealing password etc.). The area overhead, however, is a few thousand gates (the minimum overhead for login attacks is 1,341 gates). Therefore, due to this overhead it can not be applied during the fabrication stage of the microprocessor and inserting the modification requires extensive access to the design internals.

A public-key encryption circuit attack has been presented in [9]. The authors attack a circuit by turning off portions of the circuit, enabling a key-leaking attack. The malicious hardware still incurs a non-negligible overhead of 406 gates, and can only applied to RSA-specific designs.

The concept of privilege escalation attacks has been extensively studied in the software domain [10]. The dramatic increase of mobile devices has resurfaced privilege escalation problems as a problem of interest: The attackers exploit programming or hardware bugs to stage privilege escalation attacks [11]. These attacks, however, target embedded processors and rely on existing bugs.

## III. Privilege mode escalation attack

In this section we discuss the prerequisites of the privileged mode escalation attack, present how this attack could be staged in a modern microprocessor, discuss potential payloads and also present the limitations of the presented attack.

### A. Attack prerequisites

Staging a privilege mode escalation attack in a modern microprocessor, through address space identifier corruption, requires:

- Distinct privilege levels, such as kernel, supervisor, user etc. Different privilege levels enhance the security of the microprocessor, as a limit is imposed on process resource usage. During normal operation, the privilege level can be modified only with permission from the operating system.
- Existence of address space identifier, as part of the instruction translation lookaside buffer (I-TLB). An address space identifier points to the virtual space allocated for a specific process, and it allows context switching without the need for TLB flush. As TLB flushes are expensive, use of address space identifiers greatly increase the performance of the microprocessor.

Almost all modern microprocessors implement the two aforementioned features. Privilege levels first appeared with the 80386 [12] in the x86 architecture, with 4 distinct privilege levels ("Rings"): Ring 0 has the highest privileges (kernel mode), while Ring 3 has the least privileges (application mode). Faults in a ring affect only rings of the same or less privileges. Similarly, the latest Alpha microprocessors also implement 4 privilege levels [13]: Kernel, executive, supervisor and user.

As for the address space identifier, the Intel Pentium Pro introduced the page global enable (PGE) flag, that can be used to avoid TLB flushes during context switch. The Alpha 21264 introduced a similar identifier, named Address Space Number (ASN), where only TLB entries that match the process ASN would be considered valid. Furthermore, both Intel and AMD (starting with the Nehalem VT-X [14] and SVM [15] respectively) implemented a similar feature, where the address space tag is built in the TLB and dedicated hardware check tags for validity. This greatly increased the performance of the x86 architecture, as TLBs are designed to operate completely in hardware, with extremely low latency. Thus, the prerequisites for the presented attack exist in the latest commercial modern microprocessors, providing a common stage of attack.

### B. Staging the attack

In this paper, we focus on attacks that can be deployed during the fabrication stage of the design. Thus, any modifications should consist of simple gate alterations or interconnection tampering, in order to introduce stuck-at, delay, coupling faults etc. As such faults could potentially create catastrophic effects during microprocessor deployment, multiple, no overhead modifications may be required in different parts of the microprocessor.

An example of an address space identifier corruption appears in Fig. 1, where the I-TLB of the Alpha 21264 microprocessor is presented. In this case, a delay fault is introduced at the most significant bit of the address space identifier (called adress space number - ASN in the Alpha). A delay fault can be introduced during fabrication by modifying the interconnection (through wire sizing) between the memory elements storing the next address space identifier, and the location where the comparison occurs. Alternatively, a buffer between the two aforementioned locations can be inserted.

Given the maliciously inserted delay fault, the most significant bit of the address space identifier may not capture the intended value during process context switch and will end up corrupted. For example, when the process to be executed has an identifier of 128 (`1000000`, given 8-bit address space identifiers), then a delay fault will convert the identifier to 0 `00000000`, process ID 0 is reserved for kernel processes). Therefore, the next instruction of the modification-aware process will use a different address space than the one allocated by the operating system (Section III-D discusses the potential limitations of the attack). Thus, if the next instruction is a request for elevated access, or a carefully crafted memory store instruction that modifies the kernel structures, then a modification-aware process can access and/or write to unauthorized memory address space.

### C. Potential payloads

Once the process can execute an instruction in an unauthorized memory address space, there are different ways to escalate the process privileges. We should note that the options here are heavily instruction-set and architecture dependent; thus, the attacker should have some knowledge about the maliciously modified platform.

The most straightforward way to escalate privileges is to execute an instruction that serves this purpose. Since the microprocessor address space is momentarily corrupted and matches that the process to a different address space, it would

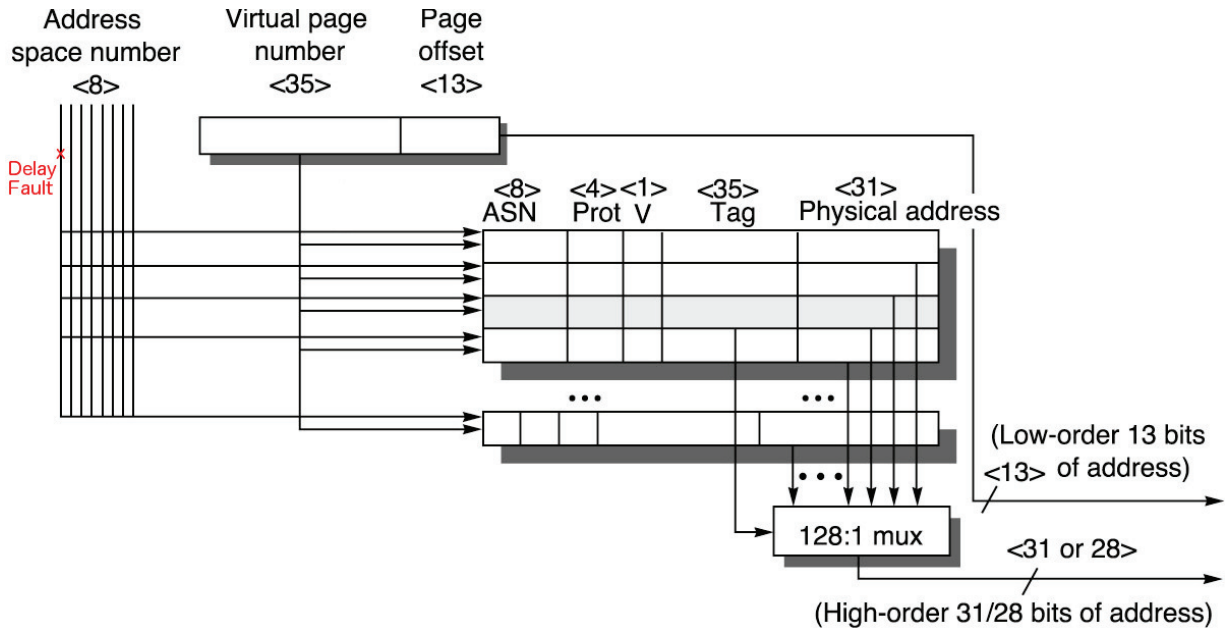


Fig. 1. Alpha 21264 memory structure [16]

grant access to the resources belonging to the process under attack. If this process is a kernel process, then the attacker can get kernel privileges and compromise the system.

However, not all instruction set architectures include instructions that directly elevate privileges. Privilege escalation usually occurs in an as-needed basis. Therefore, another way to attack the system to gain escalated privileges is to write directly to the kernel structures. These unauthorized accesses alter the privileges of the attacking process, masking it as a kernel process or altering the allocated virtual address space. This type of payload requires extensive knowledge of the operating system kernel, as well as dynamic information of the attacking process.

Finally, another way to escalate privileges is to overwrite locations where there is a priori known information. Examples of such information include interrupt handlers, shared libraries and operating system specific code. This payload requires information about the kernel, but it is independent of where the attacking process is located in memory. Because of this flexibility, we will use this type of payload to gain escalated privileges, as presented in Section IV.

### D. Attack limitations

With no-overhead malicious modifications, comes very limited control of the attack. Specifically, there are a number of conditions that need to be satisfied in order to be able to successfully deliver a privilege escalation attack:

1) The attacking process must receive a specific process ID.
2) When the address space identifier is corrupted, the "new" identifier must belong to a process with escalated privileges.
3) Context switch should occur directly before that one instruction that will attack the system.

Therefore, the attack will take place in a non-deterministic time. Experimental results presented in Section V corroborate that most of the attacks can go through within a few thousand cycles.

A problem with the presented attack is that modification-unaware processes that access the memory directly after context switch may end up with corrupted memory contents. However, this is rare, as context switch usually happens during I/O access and in modern operating systems process usually do not use the whole quantum [17]. Indeed, experimental results presented in Section V show that the workload used was not corrupted. However, as this could eventually lead to problems in other configurations and operating systems, we are currently working on masking the effect of different address space identifiers in modification-unaware processes.

## IV. Experimental Setup

In this section we discuss the specific parameters of the presented work.

### A. Microprocessor model

The test vehicle for our study is the Alpha 21264 microprocessor. The block diagram of the Alpha 21264 appears on Fig. 2. It features aggressive speculative and out-of-order execution, with a peak execution rate of six instructions per cycle. The 3-stage fetch unit can bring up to four instructions at any given cycle.

The DTLB structure of the Alpha 21264 is presented in Fig. 1. The DTLB contains 128 entries and it is fully associative. The address space number (ASN) is the address space identifier that ensures that the TLB is not flushed on a context switch. The translation begins by sending the virtual address to all tags. Once a (valid) match is identified, it goes

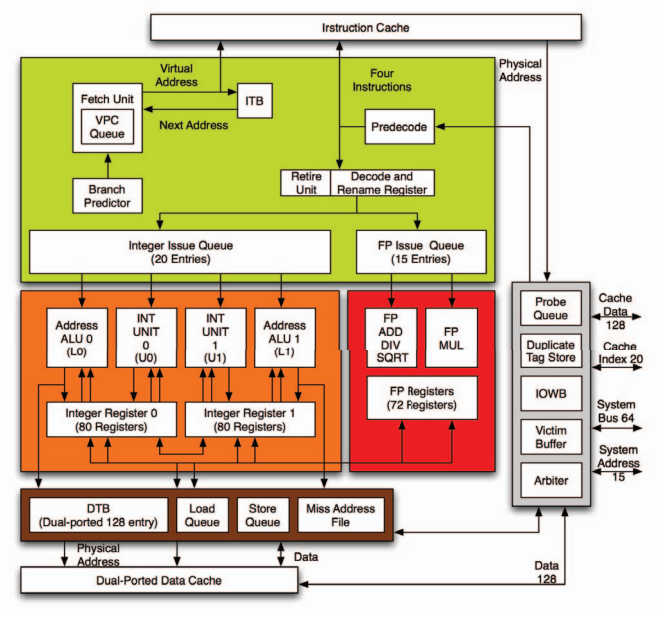


Fig. 2. Block diagram of Alpha 21264

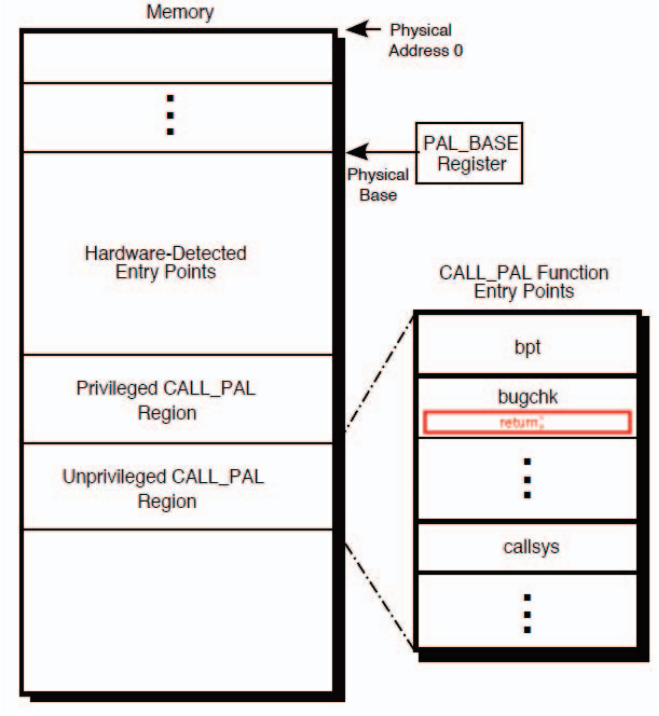


Fig. 3. Alpha 21264 PALcode entry points into memory

through a 128:1 multiplexor, augmented with the physical page frame to form the final physical address. The Alpha DTLB also contains a protection permission ("Prot" in Fig. 1) and valid ("V") bits.

### B. Attack payload

As discussed in Section III-C, once the address space identifier is corrupted the attacker can utilize different options in order to gain escalated privileges. In this paper, we modify the the operating system-specific special kernel functions (called Privileged Architecture Library code - PALcode in the Alpha instruction set architecture).

Alpha PALcode provides a hardware abstraction layer that can be used to implement hardware special functions, such as [18]:

- Instruction that require complex sequencing
- Translation buffer management (flush/load)
- Interrupt and exception dispatching etc.

The Alpha architecture lets these functions be implemented in standard machine code, that is resident in memory. Fig. 3 presents the physical memory layout of the machine with PALcode installed in a location defined by the PAL_BASE register. This register is written by the operating system, and is known a priori for each operating system. Because PALcode is resident in main memory, not all physical memory is available to the operating system code.

Since the PALcode resides in main memory, as soon as the attacking process can access the kernel memory address space, it can overwrite part of the PALcode. As only one instruction can be executed in the foreign address space, a potential attack mechanism is to abruptly return to the user process while still in kernel mode. This is presented on the right hand side of Fig. 3: The attacking process overwrites an instruction belonging to the execution area of the PALcode instruction "bugchk"; the new instruction is a simple return. Therefore, when the "bugchk" instruction is executed, the system will return to the user code, still in PALcode environment (in non-malicious code, a special instruction is used to return from PALcode). The PALcode environment has the most privileges in the system, as it has complete control of the machine state and allows all functions of the machine to be controlled. Thus, after the malicious instruction is installed after the address space identifier corruption, a simple "bugchk" instruction call suffices to gain escalated privileges.

The "bugchk" instruction was specifically chosen for two reasons:

- It is an unprivileged CALL_PAL instruction, therefore it can be invoked by any process.
- It is only used for program debugging, so it will not be executed during normal system operation. Any software attempting to execute it will receive an invalid response.

### C. Experiment flow

Gem5 [19] is the functional simulator used to implement and evaluate the presented attack. Gem5 is a modular, discrete event driven computer system simulator platform written in C++. Gem5 can execute an unmodified linux kernel, with full device support, at very high speeds.

The linux kernel used in this study is the 2.4 version, optimized for the alphaev67 platform (codename for the Alpha 21264A, a smaller version of the Alpha 21264). The linux kernel source code has been slightly modified to match the

ASN to the process IDs. This modification allows the attacking process to identify their ASN by reading the OS-assigned process ID. Ongoing work explores avoiding this modification by maintaining a ASN to process ID translation table.

Similar to the BIOS functionality in the x86 architecture, a firmware layer is needed to correctly initialize the machine state. Therefore, we use the Alpha Linux Miniloader (MILO) [20], in order to load the linux kernel. MILO builds the page tables, turns on virtual addressing, installs PALcode and initializes the linux kernel.

Two different types of workload are utilized for demonstrating privilege escalation attacks in the Alpha 21264:

- Synthetic workload, that consists of infinite loops of writing to the target PALcode instruction location (as described in Section IV-B) and executing the "bugchk" instruction that will eventually provide escalated privileges to the attacker.
- SPEC2000 benchmarks, where the attacking code described previously is injected in random locations of the algorithm. Therefore, while the SPEC benchmark executes its useful workload, an continuous privilege escalation attack is taking place in the background. Six different benchmarks, namely bzip2, mcf, gap, gzip, cc and parser, are used in this study.

The linux kernel, MILO firmware, the synthetic workload and the SPEC2000 benchmarks were cross-compiled for the Alpha using the crosstool-NG tool [21].

## V. Results and Discussion

This section discusses the timing perturbations of the presented attack for the synthetic and the SPEC2000 benchmarks. As discussed in Section III, there is no expected area or power overhead of the malicious modification.

### A. Time to gain privileges

Fig. 4 presents the number of clock cycles needed for synthetic benchmarks to get escalated privileges, for 20 different trials. For clarity, the trials are sorted in decreasing order of clock cycles. As a reminder, a synthetic benchmark only contains the necessary code to alter the PALcode code and successfully execute a "bugchk" instruction. Fig. 4 shows that if the benchmark is the only (user) process in the system, then a privilege escalation attack can take place within a few thousand cycles. As more processes are added to the system competing for the CPU (10 processes for medium workload and 100 for heavy workload), the number of clock cycles needed to successfully deliver the attack greatly increases, as the process runs less often in the microprocessor. An interesting observation though, is that adding an order of magnitude more processes (100 instead of 10) does not significantly increase the number of clock cycles before privilege escalation. This is attributed to the fact that with both medium and heavy workload there is very intensive switching, so the synthetic workload has many windows of opportunities to deliver the attack.

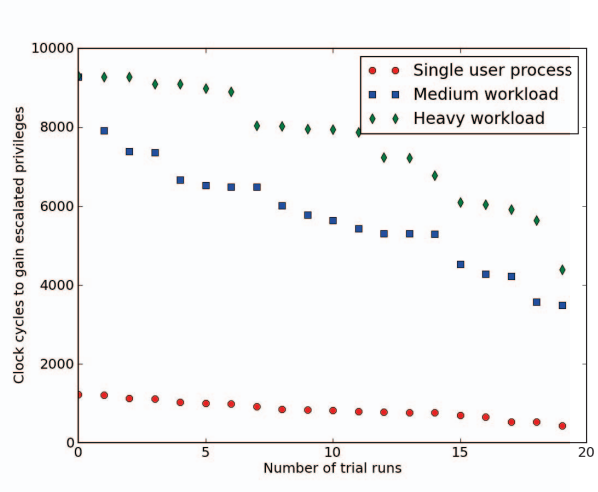


Fig. 4. Clock cycles to gain escalated privileges for synthetic benchmarks (20 trials)

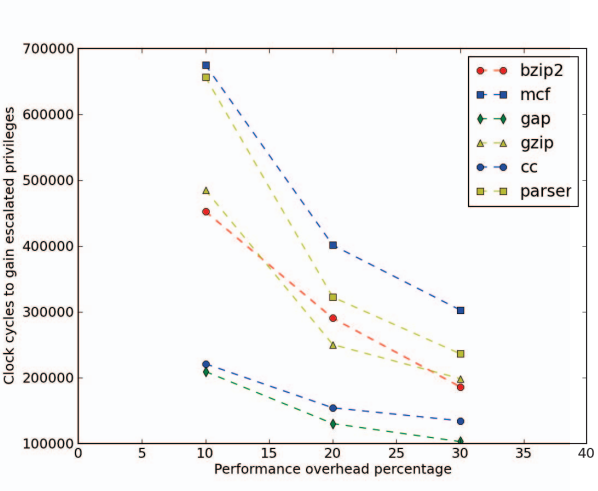


Fig. 5. Clock cycles to gain escalated privileges for SPEC2000 benchmarks (average of 20 trials), for given performance overhead

### B. SPEC benchmarks performance overhead

The number of extra instructions maliciously added to an existing benchmark affect how many clock cycles are required to receive escalated privileges. Fig. 5 presents the average number of clock cycles needed to deliver the attack for different number of performance overhead (i.e., added instructions), for 20 trials. Performance overheads of approximately 10%, 20% and 30% are examined.

As expected, there is a linear decrease of the time the attack successfully takes place for increasing performance overhead, for all given benchmarks. It should be noted that the average time-to-attack figures vary for different benchmarks; this is attributed to different numbers of I/O accesses. Since a few thousand cycles is a negligible fraction of time in real-world attack deployment, the performance overhead of the SPEC benchmarks can be sustained in very small percentages.

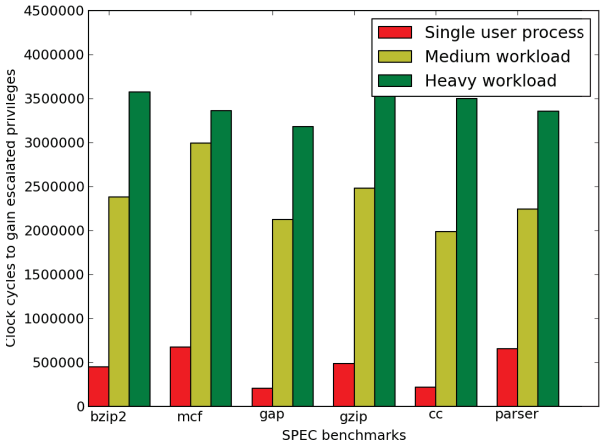


Fig. 6. Clock cycles to gain escalated privileges for SPEC2000 benchmarks (average of 20 trials), for different system activity

### C. System load impact on time-to-attack

The final set of results discusses the behavior of the attack in the presence of resource-competing processes. Fig. 6 presents the number of clock cycles required to gain escalated privileges, for the SPEC2000 benchmarks. The results are the average of 20 trial runs, for 10% performance overhead.

The first observation is that when the SPEC2000 benchmark is the only user process in the system, then a privilege escalation attack can take place within a million clock cycles. However, as more processes are added, this number increases dramatically: More processes mean less CPU time for the benchmark. Furthermore, the benchmark individual I/O demand is forcing them to yield to other processes, decreasing the probability for all 3 conditions presented in Section III-D to be satisfied at the same time. Therefore, the attack may require several million cycles to successfully deliver the payload. In our experiments, the use of typical input for the SPEC2000 benchmarks provided enough time for the attack to take place. However, the attack might not go through given small workloads.

## VI. Conclusion - Future Directions

In this paper, we presented a malicious modification that can be used during the fabrication stage of the microprocessor. Specifically, a maliciously added delay fault corrupts the address space identifier, allowing an unauthorized process to access the kernel address space, targeting escalated privileges. The attack is demonstrated on an Alpha 21264 microprocessor executing real workload. Experimental results corroborate that the attack can successfully gain escalated privileges within a few thousand cycles for synthetic workload, and within a few million cycles for maliciously modified SPEC2000 benchmarks.

Future directions of the presented research focus on increasing the controllability of the attack: Limit the catastrophic impact of the modification on modification-unaware processes, untie the ASN to process ID connection and allow the attacking process to identify kernel vs. non-kernel processes. Furthermore, potential countermeasures are currently explored, such as functional identification of the corruption or added hardware that can verify the legitimacy of the address space of the executed process.

## References

[1] U.S.D.O. Defense, "Defense science board task force on high performance microchip supply," *Washington, DC*, 2005.
[2] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware Trojans," *Computer*, vol. 43, no. 10, pp. 39–46, 2010.
[3] Y. Jin and Y. Makris, "Hardware Trojans in wireless cryptographic ICs," *IEEE Design and Test of Computers*, vol. 27, pp. 26–35, 2010.
[4] M. Tehranipoor, H. Salmani, X. Zhang, X. Wang, R. Karri, J. Rajendran, and K. Rosenfeld, "Trustworthy hardware: Trojan detection and design-for-trust challenges," *Computer*, vol. 44, no. 7, pp. 66–74, 2011.
[5] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Security analysis of logic obfuscation," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE, 2012, pp. 83–89.
[6] *http://www.poly.edu/csaw2011/csaw-embedded*.
[7] Y. Jin, M. Maniatakos, and Y. Makris, "Exposing vulnerabilities of untrusted computing platforms," in *IEEE International Conference on Computer Design*. IEEE, 2009, pp. 91–96.
[8] 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*. USENIX Association, 2008, pp. 1–8.
[9] 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*. IEEE, 2007, pp. 296–310.
[10] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proceedings of the 12th USENIX Security Symposium*, 2003, vol. 12, pp. 231–242.
[11] L. Davi, A. Dmitrienko, A.R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," *Information Security*, pp. 346–360, 2011.
[12] J. Uffenbeck et al., *The 80x86 family: design, programming, and interfacing*, Prentice Hall PTR, 1997.
[13] COMPAQ, "Alpha 21264 Microprocessor Hardware Reference Manual," Tech. Rep., Compaq computer corporation, 1999.
[14] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel virtualization technology: Hardware support for efficient processor virtualization," *Intel Technology Journal*, vol. 10, no. 3, pp. 167–177, 2006.
[15] AMD, "AMD Secure Virtual Machine Architecture Reference Manual," Tech. Rep., AMD Corporation, 1999.
[16] D.A. Patterson and J.L. Hennessy, *Computer organization and design: the hardware/software interface*, Morgan Kaufmann, 2009.
[17] J. Aas, "Understanding the linux 2.6. 8.1 cpu scheduler," *Retrieved Oct*, vol. 16, pp. 1–38, 2005.
[18] R.L. Sites, *Alpha architecture reference manual*, Digital Pr, 1998.
[19] N. Binkert et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.
[20] D.A. Rusling, "Linux on alpha axp—milo, the mini-loader," 1994.
[21] "Crosstool-NG," *http://crosstool-ng.org/*.