AppWatch: Detecting Kernel Bug for Protecting Consumer Electronics Applications

Jeaho Hwang, Jinkyu Jeong, Hwanju Kim, Jin-Soo Kim and Joonwon Lee

Abstract — *Most consumer electronics products are equipped with diverse devices since they try to provide more services following the convergence trends. Device drivers for those devices are known to cause system failures. Most previous approaches to enhance reliability have been concerned with the kernel, not with applications. In consumer electronics, however, a main application plays a core role of the product. This paper proposes a new mechanism called AppWatch to keep a consumer electronics application reliable against misbehavior of device drivers. AppWatch exploits page management mechanism of the operating system to protect the address space of the application. Since AppWatch can be implemented at a low engineering cost, it is applicable to most systems only if they have the virtual memory system. AppWatch also provides selective protection of applications so that other unprotected applications are isolated from performance loss, if any. We have tested AppWatch in a consumer electronics environment. The result shows that AppWatch effectively protects application programs at a reasonable performance overhead in most workloads, whereas data-intensive workloads show high overhead. AppWatch also protects applications with little performance interference to other unprotected applications.* [1]

Index Terms — Application Reliability, Embedded System, Device Drivers, Memory Protection

I. INTRODUCTION

Recently, the consumer electronics market has grown considerably, occupying a major portion of the embedded system products. Various types of consumer electronics device such as a digital TV are installed embedded system. [1]. In this market, requirement of the product is changing rapidly so that many companies regard the time-to-market as a crucial factor. Moreover, these embedded systems are equipped with many novel devices. Manufacturers find it hard to secure enough time to test and verify the sanity of drivers for those devices while satisfying the time-to-market constraint. At the same time, consumers are still complaining about buggy software in consumer electronics devices [2]. Although some companies try to provide software updates or to recall products in order to fix the software bug, it is not easy to recover their damaged confidence.

The device control software, called device driver, has been one of the major sources of system failures [3][4]. Most consumer electronics products have various devices such as USB controller and display, and the device drivers are embedded into the operating system to control the devices. Since many device drivers run in the same address space as the kernel at the privileged level, a buggy driver may cause catastrophic system failure by accessing unpermitted memory locations. This type of error is pathological since its occurrence is not deterministic and the symptoms are usually observable at later times [5].

Many ideas have been proposed to protect memory space from illegal accesses of faulty device drivers [6][7]. Most of them are focused on protecting kernel since application failure does not propagate to other applications or the kernel. In consumer electronics, however, reliability of a core application is as important as the kernel. In order to guarantee the primary functionality of a consumer electronics product, it is important to protect the core application software from the misbehaving device drivers. For example, in portable media player (PMP), a media player is one of the core applications of the device. If a device driver bug invades the player application's memory, it would not work correctly so that the PMP may fail to provide its expected service fully. Therefore, core applications should be protected to maintain the device's functionality.

In this paper, we propose a protection scheme to improve the reliability of core application software against device driver misbehavior. Our scheme is based on a threat model that a faulty device driver can corrupt any random location of the application's memory. AppWatch, a prototype of our scheme, protects core application software from invalid writing of device drivers to the application's memory by exploiting a memory management unit (MMU). After protecting the corruption, AppWatch prepares a report to help the debugging task.

Compared to previous work, AppWatch has three distinguished characteristics. First, AppWatch is implemented with a few lines of code in the kernel so that it can be efficiently ported to other operating systems at a low engineering cost. This is an important feature since porting a complex software solution to diverse platforms will cause huge cost. AppWatch can be ported to any operating system without a hassle. Consumer electronics developers would benefit especially from this feature.

[1] This work was partially supported by Defense Acquisition Program Administration and Agency for Defense Development under the contract.

J. Hwang, J. Jeong and H. Kim are with the Department of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Korea (e-mail: jhhwang@calab.kaist.ac.kr, jinkyu@calab.kaist.ac.kr, hjukim@calab.kaist.ac.kr)

J. Kim is with the School of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea (e-mail: jinsookim@skku.edu)

J. Lee is the corresponding author with the School of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea (e-mail: joonwon@skku.edu)

Contributed Paper
Manuscript received April 14, 2010
Current version published 06 29 2010;
Electronic version published 07 06 2010.

Second, AppWatch does not need any specific hardware support other than a standard MMU. Since an embedded system is typically equipped with an MMU to manage virtual memory, most consumer electronics devices can adopt AppWatch. It makes AppWatch even more portable.

Finally, the protection mechanism of AppWatch can be applied selectively to a specific application for efficient testing. Since AppWatch scrutinizes memory operations made only to a region that belongs to a specified application, other ones can run with almost native performance. This isolation let non-core applications run without AppWatch overhead.

The rest of this paper is organized as follows: In Section II we provide related work, and Section III describes the architecture of memory management for background. We set up the threat models and present our design and implementation of AppWatch in Section IV. Section V shows the result of evaluation. Finally we conclude and suggest future work of this paper in Section VI.

II. Related Work

Until recently, most reliability issues have been addressed in the context of the kernel since a faulty application does not hurt the whole system. With the advent of kernel extension schemes and more diverse device drivers inside the kernel, these issues gain more attention.

One of traditional and well known approach is component isolation. There are several works that have sought to isolate system components in order to enhance reliability [7]. Vino encapsulates kernel extensions using software fault isolation and uses transactions to repair kernel state after a failure [8]. Nooks architecture allows each device driver to run in its private address space in order to ensure driver isolation [6][9]. A shadow driver which is based on the Nooks subsystem provides transparent recovery from device driver faults [10].

Microkernel has dealt with the core problem of a monolithic kernel in which a kernel component shares memory with each other. Instead of a single large kernel, only a tiny kernel runs in the kernel mode, and the rest of the operating system such as device drivers and file systems run as fully isolated user-mode servers and driver processes respectively. The microkernel concept has been around for 20 years [10]-[14], and recently microkernel is noticed again due to its reliability. Minix 3 [16][17] contains much smaller kernel code and fully separated server structure. CuriOS [18] protects each component with strengthened recovery. Singularity [19] is a microkernel that is mostly written in Sing#, a type-safe language to prevent memory corruption.

Virtualization is also used for device driver isolation [7]. Each virtual machine (VM) has own address space and cannot access another VM's memory region directly. Exploiting this property, entire kernel components including device driver can be isolated. LeVasseur et al. suggested Device Driver OS on a VM environment [20]. Placing device drivers in a specific VM, a fault cannot corrupt other components because VMM guarantees isolation of each VM. Overshadow, another approach using the virtualization concept, uses shadow page table to protect an application from other codes including kernel [21]. There are two shadow page tables for each protected application in order to prevent its own data from being accessed by other components.

Though these isolation techniques efficiently prevent error propagation on commodity computer hardware, they impose significant engineering cost on implementation. The main source of this cost comes from the fact that they intend to protect a kernel object from accesses of other kernel code. Since they all reside in the same address space, virtually every memory access should be examined for its validity.

Finally, hardware-based approaches have been proposed to protect memory from invalid accesses. In Mondriaan Memory Protection (MMP) [22][23], CPU checks permission every memory reference. MMP uses a memory segment instead of a memory page and provides a permissions table which represents permission for each segment. Although this hardware approach can protect memory effectively and shows acceptable performance, most consumer electronics cannot afford to be equipped with this specialized hardware.

III. Background

This section describes how the memory management works in a commodity operating system. Most modern CPUs have similar MMU, and we only describe the detail of memory management of the x86 processor.

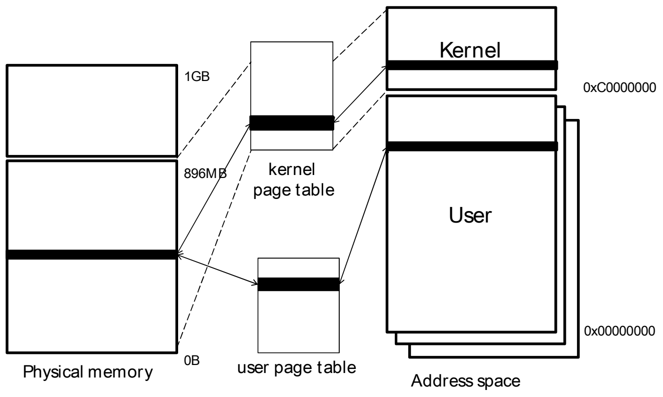


Fig. 1. Memory management system on the 32-bit x86

Figure 1 describes memory management of a commodity operating system running on 32-bit x86 architecture [24]. With 32bits address, the processor can access 4GB size of virtual memory address. There are two types of address space. First, the lowest address space is called a user address space. Usually 2~3GB is allocated to the user address space. Note that every process has its own copy of address space. Second, the highest address space is assigned for the kernel. Data and codes used in the kernel mode are stored in this area. Contrary to user address space, there is only one kernel address space, and only a privileged process can directly access this area. The kernel treats a page as the basic unit of memory management, and a virtual address is translated into a physical address via a page table.

A user page table, also called a process page table, maps a user address to a certain physical address. When a CPU references a physical page via a virtual address, it traverses the corresponding page table and checks the access privilege of the page. If the access is validated, the page is read by the CPU with the translated physical address.

The kernel also has a page table, named a kernel page table or a master kernel page global directory. Most kernel addresses are direct mapped to physical addresses by the kernel page table. For example, one of the commodity operating systems maps the linear kernel addresses 0xc0000000 through 0xf7ffffff into the physical addresses 0x00000000 through 0x37ffffff. Direct mapping enhances performance, since address translation can be simply done. CPU can access the kernel area using the same mechanism as a user page table.

Memory management system allocates an available physical page and makes a map via user page table when a process requires a page allocation. If the address of the allocated physical page is direct-mapped to the kernel virtual address, the physical page is mapped to the kernel as well as to the user space as shown in Figure 1 although the kernel virtual address is not assigned to any kernel component. In this case, accessing the kernel virtual address may intrude into the user virtual address because they share one physical address. It is one of the prevalent cases of unauthorized memory accesses from device drivers.

IV. AppWatch

In the memory management system we described, application's memory can be corrupted by a misbehaving device driver. This section describes the threat model under which a core application process can be invaded by the kernel code and our solutions to protect the application.

A. Threat model

Most embedded processors provide two mechanisms to access application's memory space. One is via MMU and the other is via DMA. Any *store* instruction in the device driver code can corrupt any part of application address space. It can easily happen by miscalculation of an address value. DMA based access, however, must be passed through DMA subsystem in the kernel. Since our target is the threat from a device driver itself, we do not consider DMA case.

1) Writing to User Address

In the kernel mode, the process has the privilege to access any memory location. A device driver is invoked either by a system call from a user application or by a hardware interrupt. Upon initiation, the device driver code may access a wrong location if there is a mistake in address calculation, and the location may be in a physical page that belongs to a user application. Then, the application may lead to an unsafe state of computation.

2) Writing Through Direct-Mapping

Direct mapping of the kernel can also cause unauthorized accesses from device drivers since physical pages are mapped both to the kernel and to the application. A buggy device driver would make a wrong access to pages which are also mapped to the application. Therefore, if the kernel address is mapped to the double mapped physical address, the writing to the kernel address affects the user address that is also mapped.

Writing through the direct mapping is especially harmful because it can affect all the processes in the system. In the former case, the kernel can only access the process which has current context. Direct mapped memory, however, could be mapped to any process's user address space so that a kernel code could corrupt a process which has no relation the current system status. It would delay the effect of the invasion and make it difficult to find and to track the bug.

B. Protection Mechanism

One of design goals in designing the solution to the aforementioned threat model is the low engineering cost since this solution is targeted for a variety of consumer electronics devices. Since most memory protection solutions call for kernel coding, a complex mechanism would incur considerable amount of engineering cost, especially for a wide spectrum of embedded platforms. Though protecting a kernel object from illegal accesses of other kernel parts needs a complex mechanism, the address space of a user application is much easier to protect since this space is isolated from the kernel space.

In order to successfully protect the user address from two threat models mentioned in the previous section with satisfying our design principle, we exploit memory management system. By setting write-protection to page table entries, AppWatch can detect wrong accesses at a negligible overhead. Also, the applications to be protected can be selected according to desired degree of protection coverage and efficiency.

Our AppWatch solution is implemented by modifying a commodity operating system that is widely employed in many embedded systems. Since most operating systems have the similar memory management system, other embedded systems can adopt it without any difficulties.

1) User Address Space Protection

Our approach named *User Address Space Protection* (UASP) is to protect the application space from writes of the kernel. Writes from the kernel code are possible only when the kernel code is invoked either by a system call or by an interrupt. In either case, when a process enters kernel mode, UASP disables write permission of the kernel to all user address spaces by modifying the permission bits of each page table entry. If a malicious kernel code tries to write to a user address space, it would be failed because no write permission is on the page table. Their write permissions are restored when the kernel finishes its task and resume the suspended user process.

The kernel needs to read from or write to a user address space in order to communicate with the application. For example, the *read* system call copies kernel memory contents which is read from a file, network, or pipe, into a user address space. Likewise, the *write* system call copies the data stored in user address space into the kernel address space. Reading from a user address space works normally since read permission is not modified. Writing to user address, however, will be denied even though it should be allowed for correct kernel operations. Therefore, it needs to enable the disabled write permissions for the target address if the writes are valid ones. The target address becomes re-protected after the end of writing.

The protection function of the UASP scheme should be placed before invoking the kernel code, and the entry point of system calls is the proper location for the function. When a system call is called, protection function is called before system call handler. If the current process is to be protected, the function traverses all the page table entries to disable write permissions. When the system call is finished write permissions are restored.

When the kernel code is running, valid writes should be allowed, and thus it needs to know which pages are allowed to be written by which functions. Several functions such as *copy_to_user* and *put_user* are used to write to user address space. To ensure that such a function runs correctly, write permissions of the pages which map the addresses to be written are allowed only during the execution of the function. The pages are re-protected at the end of the function.

2) Direct-Mapping Protection

Direct-Mapping Protection (DMP) is to protect user spaces from kernel writes when they occur through the aforementioned direct mapping. Since two virtual addresses are mapped to a same physical address, it could make confusion. If a physical page is assigned for the kernel, it cannot be mapped to a user address space. Therefore an issue arises only when a direct mapped page is mapped again to the user address space.

DMP is called when the page allocator of memory management system maps the physical address in the direct-mapped area to user address, as shown in Figure 1. After mapping is finished, the write permission of the page is disabled. If a process in the kernel tries to write to the protected kernel virtual address, exception would occur and the writing would be blocked. The write permission is restored after the user virtual address is unmapped.

Implementing DMP for a commodity operating system can be done by adding of a few lines of code in the page allocation and free function. When an application requests a page mapping, the kernel allocates a physical page. If the physical page is located in the direct-mapped memory zone, DMP finds kernel virtual address which is mapped to the physical address. Then, the page table entry to which this address belongs is deprived of write permission. After the allocated page is freed, the write permission of the kernel page table entry is reinstated.

For a kernel page, Page Size Extension (PSE) may be applied to when a larger size of page is needed. When it is set on a page table entry, this entry represents much larger size of a page [25]. Since the number of page table entries gets smaller with this feature, the hit ratio of the Translation Lookaside Buffer (TLB) increases, and so does the performance. PSE complicates our DMP scheme since protecting a small user page which is mapped to a large kernel page would incur unnecessary blocking of kernel writes. If a large user page is mapped to a small kernel page, some part of the user page would be left unprotected. Therefore, it is necessary for the DMP scheme to prohibit the use of PSE flags. Since our schemes are for debugging during software development, the performance loss of the finalized software will not be affected.

C. Discussion

Additional 400 lines of the kernel code were developed for UASP, and 200 lines for the DMP part. UASP needs more work since all the functions of the invoked kernel code should be examined to decide which pages are updated. For DMP it needs to examine only when a page in the direct mapped region is allocated.

Compared to Nooks, a kernel protection approach based on fault isolation consisted of about 22,000 lines of code [6], the coding overhead of AppWatch is negligible. Smaller code size usually means less engineering cost and better reliability, hence better software quality.

Furthermore, the kinds of operating systems for embedded systems are so diverse that porting a scheme to various platforms incur tremendous cost. In this sense, AppWatch boasts its low engineering cost for real world deployment. In summary, AppWatch shows better reliability and portability than other large-size techniques. In consumer electronics market, these two benefits make easy to develop and deploy a device.

Most embedded systems are equipped with limited computing power, and thus any tool like AppWatch, Nooks [6], or Overshadow [21] should avoid spending too much computing power if it is to be used for embedded systems. Unlike the rest of them, AppWatch can be removed from the kernel for final products since it does not change any original functions of the kernel. Since it is expected to be used only during the debugging stage, it would not cause any waste of computing power for the final product.

V. Evaluation

When the initial idea was incubated, we were concerned with the protection coverage of our schemes. To answer this issue, a fault model was set up to inject artificial faults to device drivers and to measure how much can be detected by our schemes. Also, though performance issue is not an important issue it needs to be assessed since a significant overhead even in the debugging phase would be a burden for a tool like AppWatch.

AppWatch is implemented on Netbook which consists of 1.6GHz CPU, 1 GB RAM and 12GB SSD. Although Netbook is hardly classified as a consumer electronics device, it can be used to test consumer electronics environment because the processor is developed for a small mobile device as well as a laptop.

A. Fault Detection

To test the coverage of our schemes, a new fault injection model was set up since other previous models [6][26] corrupts random locations of the kernel. Though it is valid to test protection schemes for the kernel, it would not generate enough faults in user applications. Our fault injection tool corrupts application's memory through either user address directly or direct-mapped area. Since the target address to be corrupted is randomly chosen, the behavior of a target application can be varied.

TABLE I
FAULT INJECTION ON APPWATCH

Symptom	Writing to user address		Writing through direct mapping	
	# runs	AppWach Protected	# runs	AppWach protected
none	408	408 (all)	879	879 (all)
crash	531	531 (all)	48	48 (all)
corruption	61	61 (all)	73	73 (all)

The tool injects 1000 faults into the *tiffmedian* workload which is a part of mibench [27]. TABLE I shows the evaluation results using the fault injection tool. *None* indicates that the workload finishes normally and generates a correct output. Even a corrupted application may finish without any problem since a corrupted data may not affect the execution of the application program. *Crash* represents that the workload terminates with a fault. This situation would happen when the code area or pointer variable is corrupted. The last type of result, *corruption* represents that the workload finishes normally but generates an incorrect output. Contrary to *none*, this result indicates that the corrupted data affects the execution of the process.

As shown in TABLE I, two protection schemes of AppWatch correctly protect an application from injected faults without any false positive. AppWatch can protect all corruptions because every memory access is done through page table, and the page table entry of the protected area is write-disabled.

B. Overhead

Since one of our design principles is low cost, the overhead for executing AppWatch needs to be evaluated. Execution time of each our scheme is compared with native execution time of the workload.

1) Unit Overhead

Since UASP executes at every system call invocation, we measured execution time [28] of a simple system call *getpid()*. As UASP disables all write permission of application's address space, it sweeps all the page table entries of the application. Accordingly, the execution time of *getpid()* is proportional to the size of application's address space. Figure 2 shows the execution time of calling *getpid()* system call in the UASP kernel. The figure has revealed that the overhead of UASP is proportional to the size of address space of the application. When the address space size is 20MB, which is large enough for consumer electronics devices, one system call invocation requires only 0.5ms of additional execution time.

In order to figure out unit overhead of DMP, the execution time of the *malloc()* function is measured. Note that the *malloc()* requires memory page allocation, and thus it invokes DMP mechanism. Figure 3 shows the execution time of calling *malloc()* in the DMP kernel, with varying the request size. Compared to the overhead of UASP, the overhead is about 0.02ms constant. The unit overhead is unrelated to the requested memory allocation size since every *malloc()* operation provides one page to the application. The rest of pages are provided by demand paging facility which is common in modern operating systems. Moreover, invocation of DMP is less frequent than UASP since DMP works only one time for each page mapping.

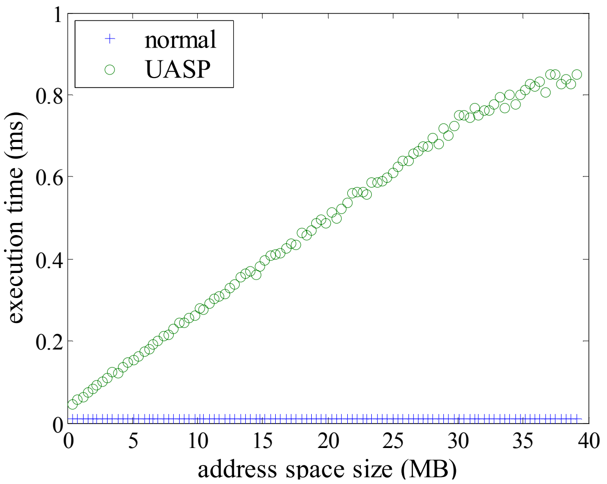


Fig. 2. Execution time of getpid system call in the UASP kernel

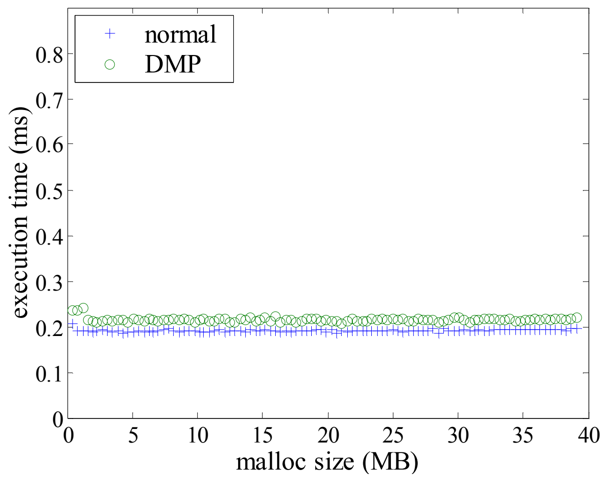


Fig. 3. Execution time of malloc function in the DMP kernel

2) Read/write overhead

As mentioned earlier, UASP causes more performance overhead. Main overhead of UASP comprises page table traverse and TLB flush. When the protection works, the protection function traverses all the page table entries and marks them as write-protected. Accordingly, the memory size of an application would affect the overhead.

In addition, temporally removing protection for the *copy_to_user* function also causes overhead since it entails an additional traverse and permission change. *Copy_from_user*, on the other hand, does not need to temporally unprotect any page because it does not write to user space. Since read and write system call are representative system calls using *copy_to_user* and *copy_from_user* functions respectively, the effect of *copy_to_user* function can be inferred by comparing the execution time of *read* and *write* system calls.

In order to figure out the performance effect of each component of UASP, we measured the execution time of *read* and *write* system call with varying the request size. We ran the test in three kernels: *normal*, *UASP* and *without write protection*, a modified *UASP* kernel which only traverses the page table so that we can observe the performance without the effect of TLB invalidation.

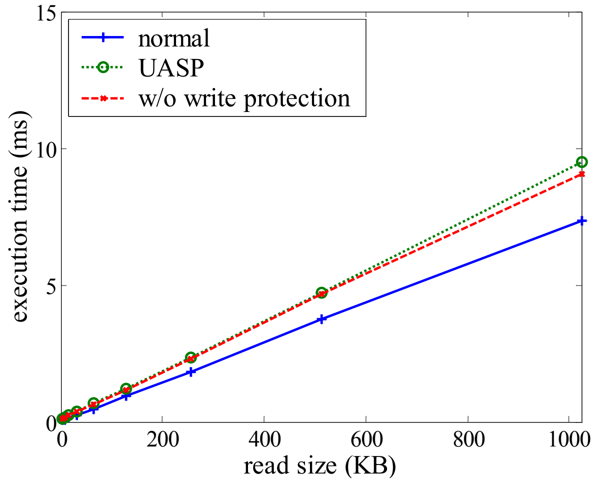


Fig. 4. Execution time of read system call

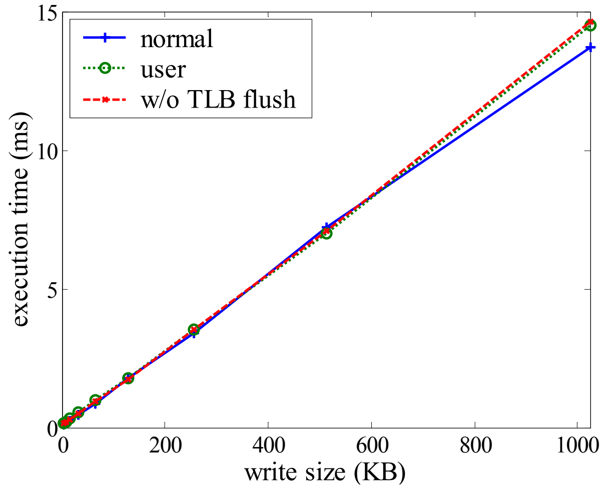


Fig. 5. Execution time of write system call

Figure 4 and 5 show the execution times of *read* and *write* system calls. At first comparing *normal* and UASP, the overhead is proportional to read and write size. Larger request size means more pages need to be protected, hence longer time for our schemes.

The overhead of *without TLB flush* is mostly caused from the page table traverse. Since the *without TLB flush* kernel does not flush TLB entries, a gap between *user* and *without TLB flush* indicates the overhead of TLB misses. This reveals that page table traverse causes more overhead than TLB miss does.

Comparing the overhead of two graphs, *read* system call presents more overhead than *write* system call due to the *copy_to_user* function. The graphs show that *copy_to_user* function is the largest source of the overhead. Therefore it can be expected that a workload which reads large size data would present low performance with UASP.

3) *Overhead from benchmark*

Other than the unit performance measurement, real workload is used to evaluate how much overhead is shown in real consumer electronics environment. The mibench [27] is a well-known open-source benchmark for embedded systems. The workloads in automotive, consumer and office category were selected because they are common workload in embedded systems. The automotive category mainly consists of CPU-intensive jobs, and the other two categories represent application software for consumer electronics such as PDA.

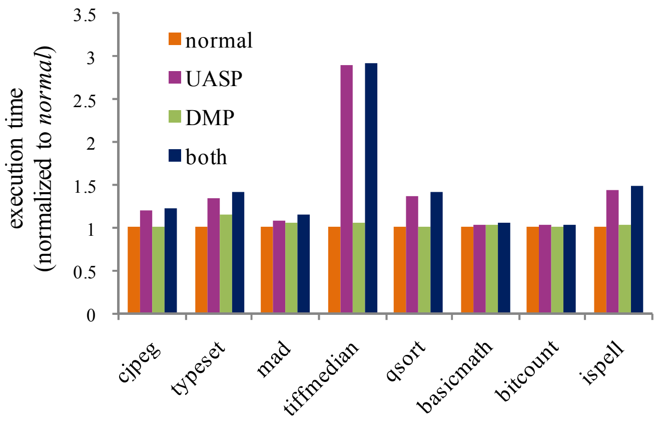


Fig. 6. Execution time of mibench workloads

Figure 6 shows the execution time of normal kernel and three types of the AppWatch kernel. The execution times are normalized to the execution time of normal kernel. UASP, in most cases, shows less than 43% overhead. Especially automotive workloads such as *basicmath* and *bitcount* show almost zero overhead since CPU-intensive job is not affected by AppWatch as expected. On the other hand, the *tiffmedian* workload, an image modifying tool, shows about 200% overhead since it reads much larger data (54.4MB) than the other applications.

In comparison with UASP, DMP shows much less overhead, maximum 14% in Figure 6. DMP runs only when a page is allocated and freed, and thus it is less frequently invoked than system calls. Therefore direct mapping protection makes little performance degradation.

TABLE II
EXECUTION TIME OF WORKLOAD RUNNING WITH PROTECTED / UNPROTECTED PROCESS

Status of background process	Execution time (s)
Running with unprotected application	1.069
Running with protected application	1.086

In order to figure out how much overhead the workload shows when a selectively protected application is running in background, we evaluate the performance of two processes of the *tiffmedian* workload running at the same time. We run the processes both in *normal* kernel and in UASP kernel but only one process is selectively protected.

As shown in TABLE II, the execution time of the process running with protected process shows about 1.5% performance degradation compared with running with non protected process. The result proves that AppWatch can selectively protect application software with little performance interference to others.

VI. Conclusion

AppWatch is designed to protect core application programs equipped in a consumer electronics device from bugs in device drivers. Since it is based on the page protection mechanism that is widely available in most operating systems, it is much more efficient than the other memory protection schemes. Also, it does not cause any modification on kernel behavior resulting in high portability at an extremely low engineering cost. Our experiments show that AppWatch protects core applications without fault negatives and with acceptable performance overhead in most cases. Even though it is not an important issue, if the performance overhead is a concern, the overhead incurred for traversing a page table can be reduced using a shadow page table as Overshadow [25] suggested.

References

[1] S, Moon, J, Kim, K, Bae, J. Lee and D. Seo, "Embedded Linux implementation on a commercial digital TV system," *Consumer Electronics, IEEE Transactions on* , vol.49, no.4, pp. 1402-1407, Nov. 2003

[2] L, Cauley and M. Kessler, "Dropped calls plague iPhone 3G, and not just in U.S.," USA TODAY, August 17, 2008,

[3] V. Orgovan and M. Tricker, "An introduction to driver quality," *Microsoft WinHec Presentation DDT301*, Redmond, WA, August 2003.

[4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles,* pp. 73-88, Banff, Alberta, Canada, October 21 - 24, 2001.

[5] R. Hastings and B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," In *Proceedings of the Winter Usenix Conference*, pp. 1-12, January 1992.

[6] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles,* pp. 207-222, Bolton Landing, NY, USA, October 19 - 22, 2003.

[7] A. Tanenbaum, J. Herder, and H. Bos, "Can we make operating systems reliable and secure?" *Computer*, 39(5):44–51, 2006.

[8] M. I. Seltzer, Y. Endo, C. Small, K. A. Smith, "Dealing with disaster, surviving misbehaved kernel extensions," In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, pp. 213-227, Seattle, Washington, United States, October 29-November 01, 1996.

[9] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers 2002. "Nooks: an architecture for reliable device drivers," In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop,* pp. 102-107, Saint-Emilion, France, July 01 - 01, 2002.

[10] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," *ACM Transactions on Computer System,* pp. 333-360, 24, 4 (Nov. 2006).

[11] J. Liedtke, "On micro-kernel construction." In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles,* Copper Mountain, Colorado, United States, December 03 - 06, 1995.

[12] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. "Mach, A new kernel foundation for UNiX development," in *Proceedings of the Summer 1986 USENIX Conference*, pp. 93-112, July 1986.

[13] D.R. Cheriton, "The V Kernel: A Software Base for Distributed Systems," *IEEE Software,* vol. 1, no.2, pp. 19-42, 1984

[14] A. Bricker, "A new look at micro-kernel-based UNIX operating systems: Lessons in performance and compatibility," In *Proceedings of EurOpen Spring'91 Conference*, Tromsoe, Norway, May 1991.

[15] S. J. Mullender, G. van Rossum, A. S. Tananbaum, R. van Renesse, and H. van Staveren, "Amoeba: a distributed operating system for the 1990s," *Computer* , vol.23, no.5, pp.44-53, May 1990.

[16] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Modula System Programming in MINIX 3," *USENIX ;login*, April, 2006.

[17] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "MINIX 3: a highly r eliable, self-repairing operating system," *SIGOPS Operaing System Rev.* 40, 3, PP. 80-89, July 2006.

[18] F. M. David, E. M. Chan, J. C. Carlyle and R. H. Campbell, "CuriOS: Improving Reliability through Operating System Structure," In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, Dec, 2008.

[19] G. C. Hunt et al, "An overview of the Singularity project," In *Tech. Rep. MSR-TR-2005-135*, Microsoft Research, October 2005.

[20] J. LeVasseur, V. Uhlig, J. Stoess, S. Götz, "Unmodified device driver reuse and improved system dependability via virtual machines," In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, p.2-2, December 06-08, 2004, San Francisco, CA.

[21] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems." In *Proceedings of the 13th international Conference on Architectural Support For Programming Languages and Operating Systems,* pp. 2-13, Seattle, WA, USA, March 01 - 05, 2008.

[22] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," In *Proceedings of the 10th international Conference on Architectural Support For Programming Languages and Operating Systems,* pp. 304-316, San Jose, California, October 05 - 09, 2002.

[23] E. Witchel, J. Rhee, and K. Asanović, "Mondrix: memory isolation for linux using mondriaan memory protection," In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles,* pp. 31-44, Brighton, United Kingdom, October 23 - 26, 2005

[24] R. Love, "Linux Kernel Development, 2nd ed.," Novel Press: Indianapolis, 2005, pp. 181~194.

[25] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual", August 2008.

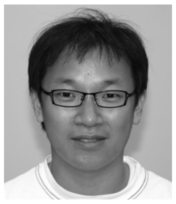
[26] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The Rio file cache: surviving operating system crashes," In *Proceedings of the Seventh international Conference on Architectural Support For Programming Languages and Operating Systems,* pp. 74-83, Cambridge, Massachusetts, United States, October 01 - 04, 1996.

[27] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," In *Proceedings of Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on* , vol., no., pp. 3-14, 2 Dec. 2001.

[28] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE* 93 (2), 216–231 (2005). Invited paper, Special Issue on Program Generation, Optimization, and Platform Adaptation

Biographies

Jeaho Hwang received his BS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2007, and MS degree in computer science from KAIST in 2009. He is currently a PhD candidate in the Computer Science Department, KAIST. His current research interests include operating systems, system reliability, virtualization and embedded systems.



Jinkyu Jeong received his BS degree from the Computer Science Department, Yonsei University, and the MS degree in computer science from the Korea Institute of Science and Technology (KAIST). He is currently a PhD candidate in the Computer Science Department, KAIST. His current research interests include real-time system, operating systems, virtualization and embedded systems.

Hwanju Kim received his BS degree in information and computer engineering from Ajou University, Korea, in 2006, and MS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2008. He is a PhD in the Computer Science Department, KAIST. His research interests include virtual machine, embedded system, and storage system.

Jin-Soo Kim (M'89) received his BS, MS, and PhD degrees in Computer Engineering from Seoul National University, Korea, in 1991, 1993, and 1999, respectively. He is currently an associate professor in Sungkyunkwan University. Before joining Sungkyunkwan University, he was an associate professor in Korea Advanced Institute of Science and Technology (KAIST) from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002 as a senior member of research staff, and with the IBM T. J. Watson Research Center as an academic visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems.

Joonwon Lee received his BS degree in computer science from Seoul National University in 1983 and the MS and PhD degrees from the Georgia Institute of Technology in 1990 and 1991, respectively. He is currently a professor in Sungkyunkwan University. Before joining Sungkyunkwan University, he was a professor at the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea from 1992 to 2008. His current research interests include low power embedded systems, system software, and virtual machines.