Kefence: An Electric Fence for Kernel Buffers*

Nikolai Joukov, Aditya Kashyap, Gopalan Sivathanu, and Erez Zadok
Stony Brook University
Computer Science Department
Stony Brook, NY 11794-4400
{kolya,aditya,gopalan,ezk}@cs.sunysb.edu

ABSTRACT

Improper access of data buffers is one of the most common errors in programs written in assembler, C, C++, and several other languages. Existing programs and OSs frequently access the data beyond the allocated buffers or access buffers that were already freed. Such programs and OSs may run for years before their problems can be detected because improper memory accesses frequently result in a silent data corruption. Not surprisingly, most computer worms exploit buffer overflow errors to gain complete control over computer systems. Only after recent worm epidemics, did code developers begin to realize the scale of the problem and the number of potential memory-access violations in existing code.

Due to the syntax and flexibility of many programming languages, memory access violation problems cannot be detected at compile time. Tools that verify correctness before every memory access impose unacceptably high overheads. As a result, most of the developed techniques focus on preventing the hijacking of control by hackers and worms due to stack overflows. Consequently, hidden data corruption is given less attention.

Memory access violations can be efficiently detected using the hardware support of the paging and virtual memory. Kefence is the general run-time solution we developed that allows to detect and avoid in-kernel overflow, underflow, and stale access problems for internal kernel buffers. Kefence is especially applicable to file system code because file systems operate at a high level of abstraction and require no direct access to the physical memory. At the same time, file systems use a large number of kernel buffers and file system errors are most harmful for users because users' persistent data can be corrupted.

Categories and Subject Descriptors

D.4.5 [**Software**]: Operating Systems—*Reliability*

*This work was partially made possible by NSF CAREER EIA-0133589 and CCR-0310493 awards and HP/Intel gifts numbers 87128 and 88415.1.

General Terms

Reliability, Security, Design

Keywords

Security, Buffer overflow, File systems

1. INTRODUCTION

Improper memory accesses are a common software problem. Erroneous buffer accesses include accesses outside of the buffer (buffer overflow or underflow) and accesses to data buffers that were already freed (stale data accesses). Invalid data writes can corrupt the data stored in some other buffer—either adjacent to the intended buffer or a completely different one, allocated in the same place of the freed buffer. On read, such accesses can furnish the executing process with wrong data that is not part of the legitimate buffer and thus corrupt and affect some related information later on. Many of these problems go unnoticed or result in hidden data corruption and can stay undetected for long periods of time.

Only after recent outbreaks of many computer worm epidemics did programmers begin to realize how widespread these bugs are. Indeed, the vast majority of the existing worms use stack overrun vulnerabilities to gain complete control over the computer systems.

Unfortunately, compile time solutions cannot detect all memory-access–related problems whereas all existing run-time solutions have non-negligible overheads. As a result, several solutions were proposed that either randomize the memory image or hide the return address pointer stored on the stack. Hardware developers started to include mechanisms that prevent the execution of code in stack memory [6]. All these methods prevent control hijacking but do not help detect or prevent the in-memory data corruption.

Most modern processors support paging and virtual memory. A kernel exception is generated once a virtual page that is not mapped to a physical one is accessed. Such a guard page can be aligned after or before a buffer to detect buffer overflow or underflow errors. In addition, a guard page can be used to detect accesses to stale memory buffers that were already freed. This method trades virtual address space for CPU cycles. In fact, this method allows checking all memory accesses with no CPU overheads.

User mode libraries that protect user buffers with guard pages are available for a number of OSs. For example, the `libgmalloc` library [8] and the `ElectricFence` malloc debugger [16] can be linked with an application and used to

Table 1: Usage of `kmallocs` and `vmallocs` in Linux 2.6.11.7 kernel.

	vmalloc	**kmalloc**
Total calls in the kernel code	505	4,469
Total calls in file system code	63	748
Invoked during boot up	68	134,223
Buffers still in memory after boot	7	3,827

detect heap and stack buffer access violations. Some OSs themselves support certain forms of guard page protection too. However, only large, specially allocated buffers are protected that way. For example, in Linux only the buffers allocated using the `vmalloc` function are protected. However, the vast majority of buffers are allocated using the `kmalloc` function as shown in Table 1. Moreover, only underflow events can be detected even for the buffers allocated with the `vmalloc` function. All this leaves most of the memory-related problems undetected and renders the existing invalid memory access detection functionality useless.

We have designed a kernel tool to detect invalid memory accesses, which we call *Kefence*. It can detect invalid memory accesses with negligible overheads, protecting most of the kernel buffers from corruption and isolating kernel processes from the influence of the wrong data reads.

Guard page protection is a compromise between memory consumption and run-time overheads. Therefore, it is usually impractical to instrument the whole OS using Kefence. However, certain data-critical components can always be protected with minimal memory and CPU overheads. Thus, Kefence is especially applicable to file systems because: (1) their errors are likely to result in the corruption of the real persistent data; (2) they use a large number of memory buffers, so manual verification is difficult and error prone; (3) they use only the buffers which can be allocated using `vmalloc` which is not the case, for example, for kernel components that use DMA.

Moreover, the tremendous flexibility of the Linux *Virtual File System* (VFS) [20] has made it a popular choice for development of new file systems, and for porting existing file systems from many other Unix and Windows systems. For example, Linux 2.6.11.7 supports 53 different file systems, ranging from disk-based ones (Ext2, Ext3, Reiserfs, XFS, UFS/FFS, and more), to network file systems (NFS, SMB/CIFS, NCPFS), to distributed ones (e.g., Coda), and many more specialized ones (`/proc`, `/dev`, debugfs, and more). These file systems total 485,158 lines of complex code, out of 2,997,507 lines of code in the entire Linux 2.6.11.7 kernel (not counting device drivers). In addition, many file systems are developed and maintained outside the Linux kernel [1, 2, 19]. This large variety and investment in Linux file systems makes tools like Kefence ever more important.

Our preliminary evaluation showed that even an unoptimized version of Kefence adds less than 2% elapsed time overheads and consumes less than 1% of the available memory for file systems that use buffers more intensively than an average file system does.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 outlines the design of Kefence. Section 4 describes some interesting implementation details. Section 5 presents an evaluation of the current Kefence prototype. We conclude in Section 6.

2. BACKGROUND

Nowadays, buffer overflows are the most notorious bugs. Many programs and OS kernels are reported to have them [3].

In general, complete compile time detection of wrong memory accesses is impossible. However, run time detection requires additional processing for every memory access operation. This leads to non-negligible overheads that can be as high as several times for simple systems [9]. Modern complicated systems that perform data flow analysis have overheads of about 20% [13]. Nevertheless, many modern programming languages such as Java routinely check all memory accesses, trading efficiency for reliability.

Because detection of memory buffers boundary violations is difficult, several techniques were developed to complicate the control hijacking in case of a stack overflow. For example, the function return address can be protected from overwrites by keeping it in a nonstandard location [4]. Alternatively, the whole memory image may be randomized using *Address Space Layout Randomization* (ASLR) [17]. Processors such as Sun's Sparc, Transmeta's Efficeon, 64-bit Intel, and AMD x86 provide hardware mechanisms to forbid the execution of code stored in the stack space. Thus, the AMD NX (No eXecute) bit and Intel XD (eXecute Disabled) bit [6] were added to mark certain memory areas as non-executable. However, all these techniques protect against the execution of untrusted code but not against overwriting of good data with bad data.

Using paging mechanisms to detect memory access violations is a well known technique for user level programs. The `ElectricFence` [16] and the `libgmalloc` [8] libraries can insert *guard* pages on one of the sides of allocated buffers and detect stale memory accesses. Both libraries have a flexible and convenient interface. For example, libgmalloc can be controlled by setting values of several environment variables such as `MallocGuardEdges`. The StackGuard [5] inserts a guard page in the stack to protect against stack overflows. In that case overheads are added on a per-function-call basis, but not on a per-memory-access basis. On Windows, guard pages can be created manually using the `VirtualAlloc` and `VirtualProtect` functions [18]. However, all the described libraries provide no information in case of a detected failure. Once a page fault is generated, the corresponding program must be inspected using a debugger.

OSs usually have small stacks and allocate buffers using special functions. These memory-allocation functions are usually divided into functions for fast allocation of small memory buffers and relatively slow functions for the allocation of larger contiguous areas of virtual memory. The Linux kernel has two functions that serve this purpose: `kmalloc` and `vmalloc`. The `kmalloc` function allocates physically contiguous memory which is not swappable. The `vmalloc` function allocates memory that is contiguous in the virtual address space, but could potentially be physically non-contiguous, and can be swapped out.

The Linux kernel has an optional feature where each buffer allocated using `kmalloc` can be followed by a word (called a red zone) with a specific value. Whenever the buffer is freed, the kernel checks the value of the red zone; if the value is modified, it can detect that a buffer overrun has occurred. The disadvantage of this method is that it can only detect buffer overruns long after the fact, and cannot prevent them.

Most modern OSs support some form of guard page protection for functions that allocate large contiguous regions of virtual memory. The Linux kernel inserts a guard page between buffers allocated using the `vmalloc` function. Unfortunately, it aligns the guard page and the beginning of the buffers and therefore detects only underflow events [11]. FreeBSD can optionally protect buffers allocated using the in-kernel `malloc` function for overflows, underflows, and stale accesses using the *memguard* debugging component [10]. In addition, FreeBSD kernels can protect existing memory buffers by setting them into read-only state using the `memguard_guard` function. However, all these features are only enabled for buffers allocated with the `M_SUBPROC` flag set. FreeBSD version 6, for example, allocates only three buffers with this flag set. A feature called *Kernel Special Pool* is included on Windows NT 4.0 Service Pack 4. This feature can check for either overflows or underflows of virtual memory regions (pools) allocated with the `AllocatePool` function in the Windows kernel [14]. The protection can be enabled or disabled via a special registry key.

3. DESIGN

Kefence is designed to detect memory buffer access violations at the hardware level and therefore impose negligible run-time CPU overheads. Most modern CPUs support virtual memory. They generate a page fault if a virtual page that is not mapped to a physical page is accessed. Such a guard page is in fact, a *page table entry* (PTE) that is not associated with any single physical page. If the guard page is not accessed, then the whole system operates as if no runtime checking is going on. To detect the out-of-boundary accesses to a memory buffer, the buffer and the guard page are aligned together on the page boundary.

Because the alignment of the buffers to page boundaries can be done either at the beginning or at the end, Kefence cannot normally detect buffer overflows and underflows simultaneously. Kefence checks a buffer for overflow if the buffer and the guard page are aligned at the beginning of the buffer; it checks for underflows if the buffer and the guard page are aligned with the buffer's end as shown in Figure 1. Simultaneous checks for overflow and underflow conditions is only possible if the allocation is in multiples of the page size.

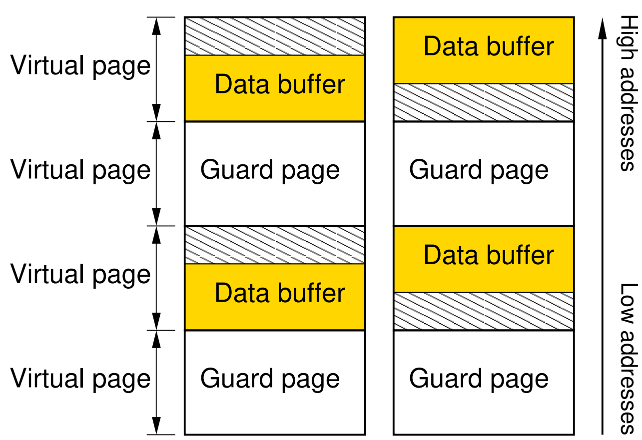

Figure 1: Buffers are aligned on the lower boundary to detect buffer underflow events (left) and on the upper boundary to detect buffer overflow events (right). The addresses increase from bottom to up.

Just-freed virtual pages become guard pages to detect stale memory accesses because they are not mapped to any physical memory anymore. Kefence marks just-freed virtual pages so that in case of a fault, these stale accesses can be distinguished from the boundary violations and analyzed.

The Linux kernel's `vmalloc` function allocates the number of requested pages (one or several) for each request. By default it aligns the buffers at the beginning and therefore it only checks the virtual buffers for underflows. Kefence supports buffer alignment on either of the buffer's sides. Kefence can align all virtual buffers on one particular side or it can decide randomly how to align a particular buffer. Random alignment allows using the same kernel to check buffers for underflows and overflows, thus increasing the chances of detecting bugs.

3.1 Reporting Problems

Buffer allocation and the related buffer boundary violations can happen in completely different parts of the kernel and can be separated by long time durations. To address this, Kefence stores extra information about every virtual buffer in the buffer's private structure. In particular, Kefence stores information about the kernel module, file name, and source code line where the buffer was allocated. We modified the page fault handler of the Linux kernel such that whenever there is an access to a guard page, it reports: (1) if the fault is caused by an overflow, underflow, or an access to a stale memory address; (2) the exact source code location of the buffer allocation; (3) the stack trace of the operation that caused the page boundary violation. In addition, all these details are logged through syslog. An example Kefence output is shown in Figure 2.

```
Buffer OVERFLOW detected!
Buffer allocated in module : wrapfs
                       file: dentry.c
                   function: wrapfs_alloc_dentry
                       line: 97
```

Figure 2: Information printed out by Kefence upon a detected buffer overflow event. The output provided is followed by a standard register dump and stack trace.

The modified page fault handler can be configured to perform various additional tasks. When security is critical, Kefence can be configured to crash the module upon a memory overflow, thereby preventing further malicious operations and further potential data corruptions. The system administrator can look at the logs to determine the location of the overflow. For critical production systems, Kefence can be configured to just log the buffer overflow without terminating the module. Kefence can auto-map a read-only or a read-write page to the guardian page table entry whenever there is an overflow. This way the code which caused the overflow can be allowed either to write or to just read the out-of-bounds memory locations. This allows critical servers to run even if a bug is detected, until a scheduled system reboot and application of patches can be afforded. Because the logs contain full information about the location and the code that caused the overflow, buffer overflows in kernel code can be diagnosed easily and multiple errors can be found. Kefence can do this in real time, making it suitable for security critical applications.

3.2 Protection Scope

As shown in Table 1, virtual memory buffers are used infrequently. Most of the time the kernel uses the `kmalloc` and `kfree` functions to allocate and free buffers. Because Kefence can only protect virtually-mapped buffers, it does not protect the buffers allocated using `kmalloc`. Therefore, to add bounds checking to the kernel code, one must use `vmalloc` instead of `kmalloc` for memory allocations. We have modified the Linux header files in such a way that this replacement is done automatically if a `-DKEFENCE` compiler option is given. This option can be set for some kernel components or for particular kernel modules.

Unfortunately, not all the `kmalloc` operations can be converted to the `vmalloc` ones. For example, drivers use DMA for data transfers and therefore require access to the physical memory. Moreover, some DMA systems can access only a small portion of the available physical memory. Also, `vmalloc` cannot be used in the context of interrupts, and in some parts of the kernel which could cause a potential deadlock while swapping virtual pages in or out. `vmalloc` consumes more virtual memory, since it allocates at least a page for each memory allocation. This is partly mitigated by the fact that modern 64-bit architectures are more widely used in enterprise settings and they make the address space a virtually inexhaustible resource. However, replacement of `kmalloc`s with `vmalloc`s results in extra consumption of physical memory because the memory is allocated in units of pages. This is an additional restriction which complicates using Kefence for the whole Linux kernel. Therefore, Kefence is mostly suitable to protect a subset of the kernel components or modules that undertake serious code modifications or are in the development stage. File systems are ideal candidates for Kefence protection because they do not require direct access to the physical memory but they still use a large enough number of memory buffers to make manual code-verification impractical.

3.3 Performance Scalability

The `vmalloc` and `vfree` functions are used infrequently. As a result, they are not optimized for performance. Every allocated buffer of virtual memory has an associated structure named `vm_struct`. This structure describes buffer's properties such as its starting address and size. All these structures are linked together in an ascending buffer-address order. Both `vmalloc` and `vfree` scan this entire list on every request. The `vmalloc` function scans the list looking for a contiguous virtual memory hole that can be used for the current allocation. The `vfree` function searches the entire linked list for the buffer with the current starting address. This is a simple and relatively efficient solution if there are only a few allocated virtual memory buffers, which is usually the case. However, the number of buffers allocated using the `kmalloc` function that are simultaneously present in memory can be as high as tens or even hundreds of thousands. Therefore, the `vmalloc` and `vfree` functions require improved scalability for Kefence purposes. A straightforward and simple solution is to use different data structures. For example, the `vfree` function can scale as $O(1)$ instead of $O(N)$ if the `vm_struct` entries are hashed based on the buffer address. The `vmalloc` function can be accelerated by caching the recently used `vm_struct` structures in a separate linked list.

4. IMPLEMENTATION

Kefence is implemented as Linux 2.4 and Linux 2.6 source patches because it requires substitution of the core kernel functions and therefore cannot be implemented as a module. Fortunately, the patch itself is relatively small because some of the required functionality is already in the Linux kernel and in most of the cases it was sufficient only to modify existing code as follows:

- we instrumented the `vmalloc` function to align the buffers on the upper boundary by default;
- we modified the `vfree` function to locate such buffers instead of raising an error;
- we modified the page fault handler to print out the details about the buffer that caused the fault such as its allocation place in the kernel source;
- a relatively large portion of the changes was related to adding new members to the `vm_struct` structure and defining new macros which replace `kmalloc` and `kfree` with `vmalloc` and `vfree` pairs to store information about memory allocation call locations in the source and control the page alignment policy of `vmalloc` for the whole kernel or only for a set of modules;
- some code was added to cache the most recently used `vm_struct` structures to speed up the `vmalloc` function and catch stale memory accesses.
- another substantial contribution, in terms of added C code size, was the hash table to speed up the `vfree` function.

The default `vmalloc` function is slow because it sequentially scans the whole linked list of `vm_struct` structures. This adds substantial CPU overheads, and it also purges the CPU data cache because every `vm_struct` structure is brought into the CPU cache. Also, the `vmalloc` and `vfree` functions call the `kmalloc` and `kfree` functions respectively, to allocate these structures. Caching the most recently used `vm_struct` structures in a separate linked list resolves these problems. No scanning or structure allocation is necessary in case of a `vm_struct` cache hit. Note that we even do not unlink the `vm_struct` structures from the default list of these structures. In addition, cached structures are not associated with any physical memory but still can be used to get the buffer allocation information in case of a stale buffer access violation. The memory consumed by our cache is small because the `vm_struct` buffers are allocated using the original `kmalloc` function. For example, a 1,024-entry cache consumes only about 64KB of memory, and improves performance for many buffers that are allocated and deallocated frequently.

To improve the scalability of the default `vfree` function we, added a hash table over the existing list of virtual memory areas. Most of the buffers in the Linux kernel are allocated using the `kmalloc` function and are smaller or equal to a page size. In addition, there is a guard page between every buffer. Therefore, our hash function can safely divide the buffer address (by shifting it) by $2 \times PAGE\_SIZE$. Our current hash table is $PAGE\_SIZE$ big, which is 4K by default on x86 architecture. Overall, the hash function that

Table 2: **Per-file counts of C lines added or changed.**

File	Lines added or changed	Added functionality
arch/i386/mm/fault.c	6	page fault handler hook
include/linux/gfp.h	1	upper or lower page edge alignment flag
include/linux/vmalloc.h	51	macros and `vm_struct` members to store extra information about the buffer
init/Kconfig	27	new kernel configuration options
mm/vmalloc.c	281	virtual memory areas cache and hash table, page fault report generation
Total:	366	

we used is

$$hash(a) = (a \gg (PAGE\_SHIFT + 1))\&\frac{PAGE\_SIZE}{sizeof(void*)}$$

which translates into $hash(a) = (a \gg 13)\&0x3ff$ for 4KB pages with 4-byte long pointers. This is a good hash function: (1) it is simple; (2) it equally distributes the load among buckets assuming that most of the allocated buffers are less than or equal to $PAGE\_SIZE$; and (3) it results in high CPU data cache hit rates because several buffers are likely to be allocated and deallocated sequentially. We resolve hash collisions with per-bucket linked lists. The average linked list size is $\frac{PAGE\_SIZE}{sizeof(void*)}$ times smaller than the one of the original `vfree`. For example, for large numbers of allocated pages, 4KB memory pages, and 4-byte long pointers, our `vfree` function is approximately 1,000 times faster than the default one.

Table 2 shows the total number of lines modified or added to implement the Kefence functionality. As we can see, a substantial portion of the changes is related to making Kefence configurable and providing as much information as possible about the boundary violations. Although, the Kefence code is architecture dependent, porting it to new architectures is trivial and requires an addition of only several lines of code.

5. EVALUATION

We evaluated Kefence on a P4 1.7GHz machine with 1GB of memory. Its system disk was a 30GB 7200 RPM Western Digital Caviar IDE and was formatted with Ext2. In addition, the machine was equipped with one more similar dedicated IDE disk formatted with Ext2 to conduct our compile benchmark.

We rebooted the machine before every benchmark run to purge file system caches and restore the memory subsystem state. We ran each test at least ten times and used the Student-t distribution to compute the 95% confidence intervals for the mean elapsed, system, user, and wait times. Wait time is the elapsed time less CPU time used and consists mostly of I/O, but process scheduling can also affect it. In each case, the half-widths of the confidence intervals were less than 5% of the mean. The test machine was running a Fedora Core 3 Linux distribution with a vanilla 2.6.11.7 kernel.

5.1 Micro-benchmarks

First we conducted a set of micro-benchmarks to measure the individual execution times of the `kmalloc`, `kfree`, `vmalloc`, and `vfree` functions. We created a kernel module that invoked these functions directly. We performed the experiments after a reboot (under the conditions described in Table 1) to avoid interference with other kernel activity. The measurement module read the contents of the i386 CPU counter register before and after the calls to the measured functions. Figure 3 shows the number of CPU cycles consumed for every function in case of cold and warm CPU caches. As we can see, the virtual memory management functions are an order of magnitude slower even if only a few virtual memory buffers are allocated. This is not surprising given that `vmalloc` calls `kmalloc` and also performs many other actions.

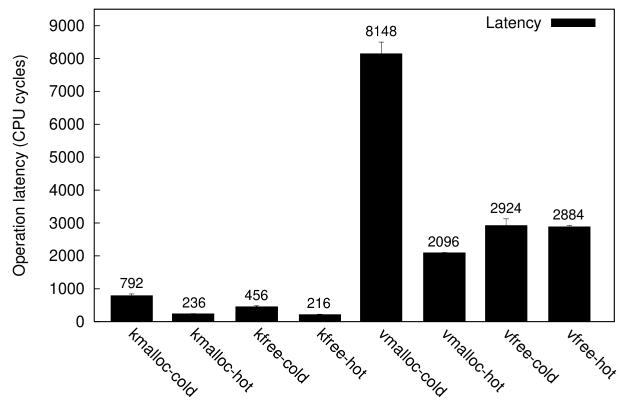

Figure 3: `kmalloc`, `kfree` and `vmalloc`, `vfree` execution times in CPU cycles with cold and hot CPU cache states with 7 pre-allocated virtual memory buffers under vanilla 2.6.11.7 Linux kernel.

`kmalloc` and `kfree` execution time is almost independent of the number of allocated buffers. However, `vmalloc` and `vfree` considerably slow down if the number of allocated buffers increases because both of them need to scan the list of allocated buffers sequentially. Even worse, list scanning requires accessing data scattered throughout memory. Not only does this result in increased scanning times due to CPU cache misses, but it also purges the CPU caches during every `vmalloc` and `vfree` operation.

Our second micro-benchmark measured the scalability of the original and our improved `vmalloc` and `vfree` functions. Figure 4 shows the dependence of the latency of these functions on the number of preallocated buffers. In particular, we sequentially allocated 5,000 buffers and then sequentially deallocated them starting from the last allocated buffer. To measure the effectiveness of our `vm_struct` structures cache, we warmed it up by running the same test before the measurement. As we can see, both the original `vfree` and `vmalloc` functions' latency grows linearly while the list of allocated memory buffers fits in the CPU cache. After

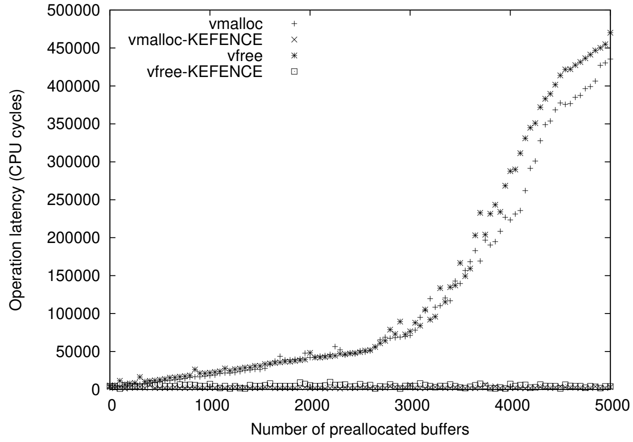

Figure 4: Dependence of `vmalloc` and `vfree` execution times on the number of already allocated buffers for the original and instrumented functions.

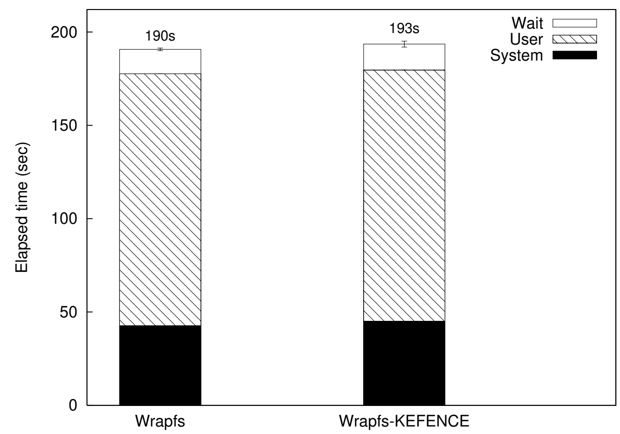

Figure 5: Am-utils compilation results over the original Wrapfs and the same file system instrumented with Kefence.

approximately 3,000 allocated buffers the list of buffers no longer fits in the CPU cache and the latency grows much faster. We can see that the hash table and the `vm_struct` cache that we added resolves the problem. In particular, the Kefence versions of `vmalloc` and `vfree` have latency that is independent of the number of allocated buffers (for `vmalloc` requests satisfied from the `vm_struct` cache). Even the `vfree` requests that miss the CPU cache result in no more than 1,024 CPU cache misses because our hash table decreases the list scanning length by 1,024 times.

5.2 Compile Benchmark

To evaluate the performance of Kefence, we instrumented a stackable file system [21] called Wrapfs. Wrapfs is a wrapper file system that just redirects the *Virtual File System* (VFS) calls to a lower-level file system. The vanilla Wrapfs uses `kmalloc`s for allocation. Each Wrapfs object (inode, file, etc.) contain private data fields which get allocated dynamically. In addition to this, temporary page buffers and strings containing file-names are also allocated dynamically. In the instrumented version of Wrapfs, we used `vmalloc` for all memory allocations so that we could exercise Kefence for all dynamically allocated buffers. Overall, we chose to instrument Wrapfs because it executes many more memory allocations than most other file systems such that Ext2; this helps demonstrate the worst-case performance of Kefence.

We compiled the Am-utils [15] package version 6.1b3 over Wrapfs (mounted over Ext2) and compared the time overhead of the instrumented version of Wrapfs with vanilla Wrapfs. Am-utils contains over 60,000 lines of C code in 430 files. The build process begins by running several hundred small configuration tests to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation: a total of 152 new files and 19 new directories. Although the Am-utils compile is CPU intensive, it contains a fair mix of file system operations. We used Tracefs [1] to measure the exact distribution of operations. The Am-utils build process uses 25% writes, 22% lseek operations, 20.5% reads, 10% open operations, 10% close operations, and the remaining 12.5% operations are a mix of readdir, lookup, etc. We used an Am-utils build benchmark because it allows us to estimate the Kefence overheads under a workload similar to these generated by advanced Linux users.

Figure 5 shows the time taken for the Am-utils compilation for vanilla Wrapfs and its instrumented version. The instrumented version of Wrapfs had elapsed time overhead of 1.4% and system time overhead of 5.9% over normal file systems. This overhead has two main causes. First, the `vmalloc` and `vfree` functions are slower than the `kmalloc` and `kfree` functions as we discussed before. Second, allocating an entire page for each memory buffer creates TLB contention which reduces performance [12].

We found that the maximum number of outstanding allocated pages during the compilation of Am-utils over the instrumented Wrapfs was 2,085 and the average size of each memory allocation was 80 bytes. That means that the vanilla version of Wrapfs consumed 166,800 bytes of physical memory. The instrumented Wrapfs on average consumed 8,540,160 bytes of physical memory (less than 1% of total physical memory on our system) and 17,080,320 bytes of virtual memory (less than 1% of available virtual memory).

5.3 I/O-Intensive Benchmark

Postmark [7] simulates the operation of electronic mail servers. It performs a series of file appends, reads, creations, and deletions. We configured Postmark to create 20,000 files, between 512–10K bytes, and perform 200,000 transactions. We selected the create vs. delete and read vs. write operations with equal probability.

Figure 6 shows the execution times of the Postmark benchmark over the Kefence-instrumented and vanilla Wrapfs. Kefence reduces the amount of available physical memory because it allocates the buffers in the units of pages. Naturally, reducing the amount of available memory reduces the amount of memory that can be used for file system caches. As a result, more I/O operations require disk accesses. We can see that in our Postmark benchmark, Kefence increased the elapsed time by 2.5 times.

We conclude that Kefence performs well for normal user workloads in terms of CPU overheads. However, I/O-intensive code that uses a lot of memory may exhaust physical or virtual memory.

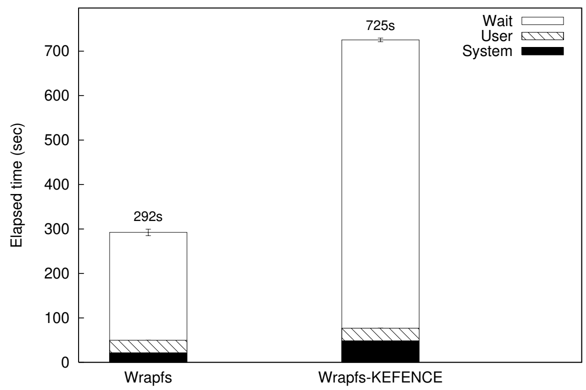

Figure 6: Postmark benchmark times for original and Kefence-instrumented Wrapfs.

6. CONCLUSIONS

We have designed, implemented, and evaluated a new in-kernel tool we call Kefence. Kefence can detect out-of-bounds and stale buffer accesses using hardware paging mechanisms. It protects most of the kernel memory buffers against corruption and prevents invalid data reads from affecting the internal kernel state. Upon detection of a memory-access violation, Kefence provides enough information to identify and resolve the problem easily.

Contrary to previous assumptions, we have demonstrated that guard-page-based protection can be applied to all memory buffers of substantial kernel components without seriously degrading performance or consuming too much memory. We have demonstrated that Kefence imposes overheads below 2% of elapsed time and below 1% of available memory for instrumented file systems under normal user workloads.

Future work. Aside from the paging-based detection of memory access violations, we are working on the kernel code compiler that generates software-based bounds checking.

7. ACKNOWLEDGMENTS

We would like to acknowledge contributions of the following people: Mohan-Krishna Channa-Reddy and Salil Gokhale developed an early Kefence prototype. Devaki Kulkarni ported Kefence to the 2.6 Linux kernel. Abhishek Rai participated in some Kefence testing. We would like to thank Charles P. Wright for useful discussions about the applicability of Kefence and for his help with the paper's preparation. Finally, we would like to thank all FSL members for their support and a productive environment.

8. REFERENCES

[1] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 129–143, San Francisco, CA, March/April 2004. USENIX Association.

[2] P. J. Braam. The Lustre Storage Architecture. `www.lustre.org/documentation.html`, October 2002.

[3] CERT Coordination Center. CERT/CC Overview incident and Vulnerability Trends Technical Report. `www.cert.org/present/cert-overview-trends`.

[4] T. Chiueh and F. Hsu. RAD: A Compile-time Solution to Buffer Overflow Attacks. In *Proceedings of the 21rst International Conference on Distributed Computing Systems (ICDCS)*, pages 409–420, Phoenix, AZ, April 2001.

[5] C. Cowan, C. Pu, D. Maier, H. Hintongif, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the Seventh USENIX Security Symposium*, pages 63–78, San Antonio, TX, January 1998.

[6] Intel. *Intel Itanium 2 Processor Reference Manual For Software Development and Optimization*. Intel Corporation, 2004.

[7] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[8] BSD Library Functions Manual. *libgmalloc(3)*.

[9] V. Markstein, J. Cocke, and P. Markstein. Optimization of Range Checking. In *Proceedings of the 17th Symposium on Compiler Construction (SIGPLAN'82)*, pages 114–119, June 1982.

[10] B. Milekic. *memguard(9)*.

[11] A. Morton. Re: [patch, 2.5] __vmalloc allocates spurious page?, October 2002. `www.uwsg.iu.edu/hypermail/linux/kernel/0210.1/2532.html`.

[12] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI '02)*, pages 89–104, Boston, MA, December 2002. USENIX Association.

[13] T. Nguyen and F. Irigoin. Efficient and Effective Array Bound Checking. *ACM Transactions on Programming Languages and Systems*, 27(3):527–570, May 2005.

[14] W. Oney. *Programming the Microsoft Windows Driver Model*. Microsoft Press, Redmond, WA, second edition, 2003.

[15] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. `www.am-utils.org`.

[16] B. Perens. *efence(3)*, April 1993.

[17] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of 11th ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, October 2004.

[18] D. A. Solomon and M. E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, Redmond, WA, 2000.

[19] M. Szeredi. Filesystem in Userspace. `http://fuse.sourceforge.net`, February 2005.

[20] E. Zadok and I. Bădulescu. A stackable file system interface for Linux. In *LinuxExpo Conference Proceedings*, pages 141–151, Raleigh, NC, May 1999.

[21] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000. USENIX Association.