SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs

Feng Qin, Shan Lu and Yuanyuan Zhou
Department of Computer Science,
University of Illinois at Urbana Champaign
{fengqin, shanlu, yyzhou}@cs.uiuc.edu

Abstract

Memory leaks and memory corruption are two major forms of software bugs that severely threaten system availability and security. According to the US-CERT Vulnerability Notes Database, 68% of all reported vulnerabilities in 2003 were caused by memory leaks or memory corruption.

Dynamic monitoring tools, such as the state-of-the-art Purify, are commonly used to detect memory leaks and memory corruption. However, most of these tools suffer from high overhead, with up to a 20 times slowdown, making them infeasible to be used for production-runs.

This paper proposes a tool called SafeMem *to detect memory leaks and memory corruption* on-the-fly *during* production-runs. *This tool does not rely on any new hardware support. Instead, it makes a novel use of existing ECC memory technology and exploits intelligent dynamic memory usage behavior analysis to detect memory leaks and corruption. We have evaluated SafeMem with seven real-world applications that contain memory leak or memory corruption bugs. SafeMem detects all tested bugs with low overhead (only 1.6%-14.4%), 2-3 orders of magnitudes smaller than Purify. Our results also show that ECC-protection is effective in pruning false positives for memory leak detection, and in reducing the amount of memory waste (by a factor of 64-74) used for memory monitoring in memory corruption detection compared to page-protection.*

1 Introduction

Memory leaks and memory corruption are two major forms of software bugs that severely threaten system availability and security. According to the US-CERT Vulnerability Notes Database [28], 39% of all reported vulnerabilities since 1991 were caused by memory leaks or memory corruption, and 55% of the most severe vulnerabilities are related to them. In the year of 2003, these two types of bugs contributed to 68% of the CERT/CC [6] advisories.

Memory leaks, caused when some allocated memory is never accessed again, can cumulatively degrade overall system performance by increasing memory paging. Even worse, they may cause programs to exhaust system resources, eventually leading to program crashes [15]. For this reason, malicious users often exploit memory leaks to launch denial-of-service attacks. Memory corruption, on the other hand, damages memory content through buffer overflow, incorrect pointer arithmetic, or other types of program errors. Similar to memory leaks, memory corruption bugs, especially buffer overflows, are commonly exploited by Internet attacks to attach malicious code through carefully-crafted input data.

There are three main approaches to address the memory leak and memory corruption problems. The first approach uses type-safe languages such as Java or the Microsoft Common Language Runtime environment [22] to eliminate the memory leak problem and reduce the chances for memory corruption. While this approach improves code quality significantly, it is not applicable to performance-critical software such as server programs. This is because type-safe languages typically introduce significant overhead, and do not allow fine-grained manipulation of data structures. As a result, most performance-critical software programs are still written in unsafe languages such as C or C++.

The second approach applies static program analysis tools, such as METAL [14], PREfix [5], Clouseau [16] and CSSV [10], to detect memory leaks and memory corruption. While these tools do not impose run-time overheads, they may miss a lot of bugs and also generate many false alarms because no accurate run-time information is available during static checks. In addition, some of these tools require annotations from the programmer, which many programmers find too tedious.

The third approach, called dynamic monitoring, is commonly used by programmers to detect memory leaks and memory corruption. Dynamic monitoring can be performed either in software or with hardware support. Purify [15]

Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11 2005)
1530-0897/05 $20.00 © 2005 **IEEE**
IEEE COMPUTER SOCIETY

is a state-of-the-art software-only dynamic tool for detecting memory leaks and memory corruption. However, Purify and most other software dynamic tools have a major limitation: incurring high run-time overhead. Sometimes these tools can slow down a program by up to 20 times [23, 7]. Therefore, they cannot be used during production runs. iWatcher [32] is a recently proposed architectural extension to reduce overheads for dynamic monitoring, but it requires new hardware extensions and therefore cannot be used in existing systems.

In this paper, we propose a low-overhead dynamic tool called SafeMem to detect memory leak and memory corruption *on-the-fly* during *production runs*. It does not require any new hardware extensions. Instead, it makes a novel use of existing Error-Correcting Code (ECC) memory technology and exploits intelligent dynamic memory usage behavior analysis to detect memory leaks and corruption. ECC-protection is used to prune false positives in memory leak detection, and to monitor illegal accesses, both to freed memory buffers and to the two ends of allocated memory buffers, to detect memory corruption. More specifically, our work has the following contributions:

- A novel use of ECC memory technology to detect memory leaks and memory corruption. Our experimental results with seven real-world buggy applications show that this method generates few false positives (0-1 for memory leak detection, and 0 for memory corruption detection), and has low overhead (only 1.6%-14.4%), 2-3 orders of magnitudes smaller than Purify. Our results also show that, compared to page protection, ECC protection can reduce the amount of memory waste by a factor of 64-74 for memory monitoring in memory corruption detection. Finally, ECC protection is also effective in pruning false positives (reduced from 2-13 to 0-1) for memory leak detection.

- A novel method that uses intelligent memory usage behavior analysis to detect memory leaks with few false positives. This method is based on a novel observation of memory object lifetime, which is validated through statistical analysis using three server programs (see Section 3).

- SafeMem, a low-overhead tool that can be used to detect memory leaks and memory corruption on-the-fly during production runs for preventing security attacks and improving software robustness.

The rest of the paper is organized as follows. Section 2 introduces the ECC memory and our novel use of this technology. Sections 3 and 4 present the methods to detect memory leaks and memory corruption, respectively, followed by the evaluation methodology in Section 5. Experiment results are presented in section 6. Section 7 discusses the related work, and section 8 concludes the paper.

2 ECC Memory

2.1 Background

Error-Correcting Code (ECC) memory is commonly used in modern systems, especially server machines, to provide error detection and correction in case of hardware memory errors. It is an extension of simple parity memory, which can detect only single-bit errors. In contrast, ECC not only detects single-bit and multi-bit errors, but it also corrects single-bit errors on the fly, transparently. Unlike parity memory, which uses a single bit to provide protection to eight bits, ECC uses larger groupings: 7 bits to protect 32 bits, or 8 bits to protect 64 bits [18]. For convenience, we call such a block of 32 bits or 64 bits an *ECC-group*. ECC requires special chipset support. When supported and enabled, ECC can function using ordinary parity memory modules; this is the standard way that most motherboards with ECC support operate. The chipset "groups" together the parity bits of memory modules into the 7 or 8-bit block needed for ECC.

Most ECC memory controllers support four modes: Disabled, Check-Only, Correct-Error and Correct-and-Scrub. In the Disabled mode, the memory controller disables all the ECC functionalities. In the Check-Only mode, the memory controller detects and reports single-bit and multi-bit errors, but it does not correct them. With the Correct-Error mode enabled, the memory controller not only detects single-bit and multi-bit errors, but it also corrects single-bit errors. This mode improves data integrity by seamlessly correcting single-bit errors. With the Correct-and-Scrub mode enabled, the memory controller not only detects and corrects errors, but it also scrubs memory periodically to check and correct hardware errors. This mode provides the highest data integrity.

ECC memory works as shown in Figure 1. At a write to memory, the memory controller encodes the involved ECC-groups using some device-specific coding algorithms. The ECC "code" (7 or 8 bits) is stored with the data in memory. At a read to memory, or during memory scrubbing, the memory controller reads the involved ECC-groups, including both data and ECC codes. It also recomputes the ECC codes based on the data just read and compares it with the stored ECC codes. If they mismatch, the memory controller automatically corrects single-bit errors, and reports multi-bit errors to the processor using an interrupt, which is delivered to the operating system.

To handle an ECC-error interrupt current operating systems, including both Linux and Microsoft Windows, simply go to the panic mode or the blue screen and report an error message to the end-user. The user has to reboot the machine to solve the problem.

1530-0897/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

(a) *Write to ECC memory* (b) *Read from ECC memory*

Figure 1: Read/Write Operations for ECC Memory

2.2 Using ECC to Monitor Memory Accesses

2.2.1 Main Idea

Our work makes a novel use of ECC memory to monitor memory accesses for software debugging. More specifically, we use ECC memory for two purposes: (1) detecting illegal accesses (e.g., out-of-bound memory accesses, or accesses to freed memory buffers) to monitored memory locations; (2) pruning false positives in memory leak detection. More details about each specific usage are described in Section 3 and 4.

Both usages require detection of accesses to some monitored memory locations. To achieve this goal, we use ECC protection in a way similar to page protection, which is commonly exploited in shared virtual memory systems [20]. Even though ECC groups are either 32 bits or 64 bits in granularity, using ECC for memory protection has to be at cache-line granularity, because accesses to main memory use this granularity.

The advantage of using ECC protection over using page protection is that the former is at cache line granularity, whereas the latter is at page granularity. Therefore, ECC protection can significantly reduce the amount of false sharing and padding space. In our experiments, we have compared these two approaches quantitatively, and our results show that ECC protection can reduce the amount of memory waste used for memory monitoring by up to 74 times (see Section 6).

These advantages of ECC protection are also exploited by some fine-grained distributed shared memory systems, such as Blizzard [25]. Different from those works, we use ECC protection for software debugging instead of implementing cache coherence operations. Therefore, we have different design trade-offs. In addition, they used special ECC memory controllers, whereas we use a standard off-the-shelf ECC memory controller, which has much more limited functionality available to software. For example, most commercial ECC memory controllers do not allow software to directly access the ECC code. Moreover, unlike page protection faults, operating systems do not deliver the ECC-error interrupt to user-level programs. Therefore, we need to first address all these challenges before we use ECC for monitoring memory accesses to watched locations.

We modify the Linux operating system to provide three new system calls: (1) *WatchMemory(address, size)*, which registers a memory region starting from *address* to be monitored by SafeMem. The memory region and its size need to be cache line aligned. (2) *DisableWatchMemory(address)*, which removes monitoring to the specified memory region. (3) *RegisterECCFaultHandler(function)*, which registers a user-level ECC fault handler. When an ECC fault occurs, the fault is delivered to this user-level handler.

In our work, we only need to detect the first access to each monitored location because: (1) For memory corruption detection, the first access to a monitored location is a bug. SafeMem then simply pauses program execution to allow programmers to attach an interactive debugger, such as gdb, to check the program state and analyze the bug. (2) For memory leak detection, the first access to a monitored location indicates a false positive. Then this location no longer needs to be monitored. Therefore, in both cases, the user-level ECC fault handler of SafeMem can disable the monitoring for the faulted lines using *DisableWatchMemory()* system call.

2.2.2 Design Issues

Data Scrambling Since most commercial ECC memory controllers do not allow software to directly modify an ECC code, we use a special trick to "scramble" the ECC code of a watched ECC-group. When *WatchMemory* is called, SafeMem first disables the ECC functionality, and writes the scrambled data into this ECC-group. It then flushes the data from cache into memory. Since ECC is disabled, the ECC code for this line remains the same, i.e., the old code. Finally, SafeMem enables ECC. Figure 2 shows the process of this trick. During the disable-enable period, we lock the memory bus to avoid any other background memory accesses, such as those made by other processors or DMAs, so that other memory locations are not affected by this *WatchMemory* operation. After this operation, the first access to this location triggers an ECC fault because of the mismatch between the old ECC code and the scrambled data.

IEEE COMPUTER SOCIETY

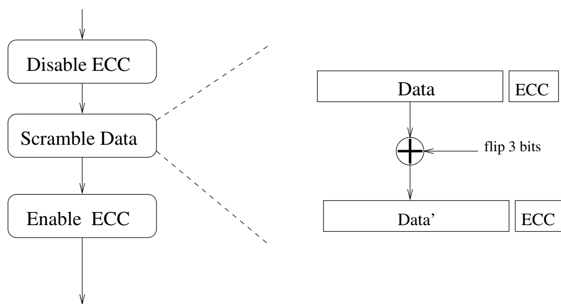


Figure 2: Implementation of *WatchMemory*

The data is not scrambled randomly. Instead, we use a special scrambling scheme to ensure two properties: (1) The scrambled data should trigger a multi-bit ECC fault instead of a single-bit error, as most ECC memory can automatically correct single bit errors without reporting to the operating system. (2) The scrambled data should have a unique signature so that it can be easily differentiated from a real hardware ECC error. In the prototype implementation of SafeMem, we flip 3 fixed bits of the original data stored in a watched line.

In addition, we also store the original data in a private memory region of SafeMem in order to differentiate an access fault from a real hardware memory error. With the original data, the SafeMem ECC fault handler can recompute the “scrambled” value and compare against the current value stored in memory. If they do not match, it is a real hardware ECC error. Otherwise, it is an access fault caused by an access to this watched location.

Differentiate Hardware Errors from Access Faults The main functionality of ECC memory is to detect memory hardware errors, which does not interfere with our techniques for two reasons. First, as we mentioned earlier, SafeMem scrambles data in a special way. When an ECC fault occurs, SafeMem first checks whether the line is monitored; if so, SafeMem checks the data to see whether it matches the scrambling signature. If yes, it is an access fault, otherwise, it is a hardware error. Second, the data stored in monitored regions is not useful because monitored regions are either padded ends or leaked buffers. Therefore, even if the data is modified because of a real hardware error, it is not critical to the program’s execution. Moreover, the original data in monitored regions is saved in SafeMem’s private memory.

Dealing with ECC Memory Scrubbing When the memory controller enables scrubbing, memory is scanned periodically to check and correct hardware errors. Therefore, special care needs to be taken in order to avoid undesired ECC faults introduced by memory scrubbing. Since most ECC memory controllers allow the OS to dynamically enable/disable scrubbing, SafeMem solves this problem by coordinating with ECC memory controllers in the following way: during scrubbing, SafeMem temporally unmonitors all the watched regions and blocks the monitored program until scrubbing finishes. Since scrubbing is infrequently performed and only during idle periods, this will not significantly affect performance. However, a better alternative would be to scrub and unmonitor the memory at page granularity, which would require changes to ECC memory controllers to signal the OS before each page scrubbing.

Dealing with Cache Effects To avoid the cache filtering effect, the *WatchMemory* operation flushes the corresponding cache line from the processor caches so that subsequent accesses to this line must access memory and therefore trigger the corresponding ECC fault. This technique also ensures that a write instruction to a watched line is also monitored (even though writes to memory do not trigger ECC checks). This is because a write to data that is not currently in cache must first load the data from memory to cache, and thereby triggers an ECC fault. After the first access is detected, the line can remain in the processor cache without being flushed because SafeMem only needs to detect the first access to a watched line.

Dealing with Page Swapping Since ECC protection is associated with physical memory, it can be affected by page swapping which changes the virtual-to-physical page mapping. A simple way to address this problem is to pin monitored pages: a page is pinned when any memory region inside is monitored, and is unpinned when it has no monitored memory regions. However, this method limits the total amount of monitored memory. To solve this problem, a better solution would be to modify the OS to unmonitor all associated memory regions when a page is swapped out, and re-monitor those regions when this page is swapped in. For simplicity, we implement the first method in SafeMem.

2.2.3 Discussion

Unfortunately, ECC has several limitations that we cannot overcome by simply using software tricks. Addressing these limitations requires hardware changes. For example, even though ECC protection is much finer grained than page protection, it is still larger than desired. In SafeMem, each dynamic buffer requires padding space of two cache lines. In addition, each dynamic buffer size needs to be cache-line aligned to avoid false sharing, which also wastes memory space. If ECC protection could be done at word granularity, such as in the Mondrian Memory Protection (MMP) [31], the amount of memory waste could be further reduced. Unfortunately, Mondrian Memory Protection still does not exist in real hardware yet.

IEEE COMPUTER SOCIETY

Some aspects of our current ECC library are device-specific. The reason is that most ECC memory controllers export a narrow, limited interface to OS. Since our study provides a strong motivation to utilize ECC for purposes other than hardware memory error detection and correction, we hope that the ECC-protection interface can be generalized to be more software-friendly, just like page protection. In other words, the interface should include the following two features: (1) An ECC memory controller allows the OS to directly modify the ECC code associated with any data. This feature is not only useful for applications like SafeMem, but also allows software to dynamically fix some transient memory errors without going to panic mode. (2) An ECC memory controller can deliver precise interrupts of ECC faults to the OS so that the OS can catch exactly the faulted instruction. Even though SafeMem does not need this feature for bug detection, this feature would allow SafeMem to enhance its functionality, such as providing programmers with precise information regarding the occurred bugs. With the above two features, SafeMem could be designed with a better hardware-software layered architecture.

3 Detecting Memory Leaks

Not all memory leaks affect software reliability and availability. *Trivial memory leaks* (leaks that only happen several times) result only in memory waste and a slight execution slowdown due to increased paging. In contrast, *continuous memory leaks* (non-stop leaking) can cause programs to run out of virtual memory and eventually crash. Crashes are especially catastrophic for long-running server programs, such as web servers, because service unavailability is directly related to loss of business. Therefore, continuous leaks are often exploited by malicious users to launch denial of service attacks.

This paper focuses on continuous leaks because they make software vulnerable. Our detection method first analyzes the run-time dynamic memory usage behavior of a program, then uses the learned behavior to detect outliers, and finally exploits ECC-protection to prune false positives.

For the convenience of description, we use the following terminology throughout this paper:

- *Memory Object:* a memory block allocated via memory allocation calls such as *malloc, realloc, calloc*, etc.
- *Live Memory Object:* a memory object that is not yet deallocated.
- *Lifetime of Memory Object:* the period from the allocation of a memory object to its deallocation.
- *Memory Object Group:* a group of memory objects. In this paper, we use a tuple $(size, callChain)$ to divide memory objects into various groups, where $size$ is the object's size, and $callChain$ is the call-stack signature[1] when the object is allocated. Even though it is possible to use other grouping methods, such as program-specific types, our experiments show that our grouping mechanism works well and does not require any semantic information from programs.

3.1 Characteristics and Classification of Continuous Memory Leaks

There are two main types of continuous memory leaks for a memory object group, and each type has different characteristics. The first type, called *always leak (ALeak)*, refers to leaks that always happen. In other words, the program does not free a group of memory objects in all possible execution paths. As a result, the number of memory objects in this group grows rapidly, and each object has an infinite lifetime. Detecting this type of memory leaks is relatively easy since it has simple characteristics.

The second type, called *sometimes leak (SLeak)*, refers to leaks that sometimes happen. In other words, in some execution paths, the program deallocates the allocated memory object, but in the other paths, the program does not free the allocated memory object. Therefore, some memory objects have finite lifetime whereas other objects of the same group have infinite lifetime. The number of leaked memory objects grows slowly, but it can still lead to memory resource exhaustion after a long period of time, resulting in program crashes. The second type is much harder to detect since the leak happens only in some execution paths.

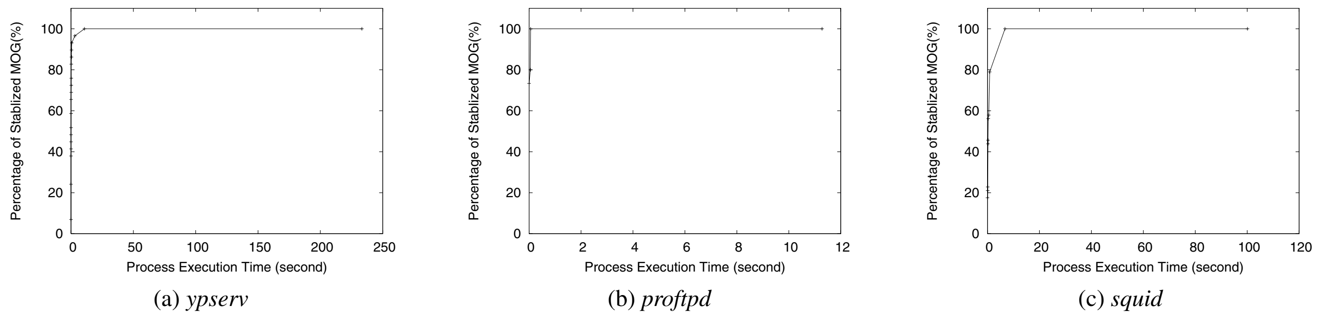
Fortunately, based on our memory usage behavior analysis using several server programs, we found that most dynamic memory objects conform to some expected lifetime. More specifically, the maximal lifetime of memory objects that belong to the same group usually remains stable after some warm-up period. Therefore, if we can dynamically capture the maximal lifetime for each object group, we can detect outliers—memory objects whose lifetime significantly exceeds the expected maximal lifetime of the corresponding object group.

Time here means the CPU time of the monitored program, which excludes time used by other running programs and time waiting for I/Os. Therefore, for server programs, *a long idle period between two consecutive client requests would not affect our detection mechanism.*

This observation is validated through statistical analysis using three server programs. To measure the stability of maximal lifetime for a memory object group, we introduce a metric called *WarmUpTime*, which denotes how long it

[1] The call-stack signature is calculated by individually applying the exclusive-OR and rotate functions to the return addresses of the most recent four functions in the current stack.

IEEE COMPUTER SOCIETY



(a) *ypserv* (b) *proftpd* (c) *squid*

Figure 3: Stability of maximal lifetime (MOG means Memory Object Group)

takes for this group's maximal lifetime to become stable. For a given memory object group, after the *WarmUpTime*, objects that belong to this group never live longer than this maximal lifetime.

Figure 3 shows the stability of maximal lifetime for three server programs: ypserv, proftpd, and squid, which are later used in our experiments to evaluate SafeMem. When we collect statistics, we use normal inputs so the memory leak bugs do not occur. Each curve on Figure 3 plots the cumulative distribution of memory object groups whose *WarmUpTime* is smaller than a given value. For example, a point (t, p) on the curve indicates that $p\%$ of the memory object groups in this program have reached the stable maximal lifetime after running for t seconds. Each memory object group is labeled by a tuple $(size, callChain)$, described in the previous subsection.

As shown in Figure 3, for all three programs, all memory object groups reach their stable maximal lifetime quickly in the very beginning of the program execution. We have also run the programs much longer, but the results remain the same. This validates our observation that the expected maximal lifetime remains stable after some short warm-up periods. Therefore, it can be used to detect potential memory leaks by dynamically monitoring each memory object's lifetime against the expected maximal lifetime associated with the corresponding object group.

3.2 Detection Process

Based on the above observation, SafeMem detects these two types of continuous memory leaks on-the-fly during production runs. The detection process includes three steps: (1) Dynamically analyze the memory usage behavior of the monitored program; (2) Detect potential memory leaks based on observed usage characteristics; (3) Use ECC protection to prune false positives.

Each of the three steps adds only a small overhead because step 1 and step 2 are performed periodically and only at memory allocation or deallocation time instead of every memory access, and step 3 is performed only for those rare memory leak suspects. The first access to a suspect disables ECC monitoring for this memory object.

3.2.1 Step1: Memory Usage Behavior Collection

For each memory object group, SafeMem dynamically collects its allocation/deallocation behavior. More specifically, SafeMem records two types of information: (1) lifetime information and (2) memory usage information. The lifetime information includes the current maximal lifetime and how long the maximal lifetime has been stable (*stableTime*). Once again, time here is measured using the CPU time.

The memory usage information includes the number of current live objects, the last allocation time, and the total memory space currently occupied by this memory object group. For each live memory object, it also records its allocation time. All live objects within the same group are linked together using a double-linked list.

At each memory allocation, the information associated with the corresponding memory object group is updated. More specifically, a new live object is added to this memory object group, and the number of current live objects is incremented by one. The last allocation time and the total memory space currently occupied by this memory object group are also updated accordingly.

Similarly, the information is also updated at each memory deallocation. First, the lifetime of the deallocated object is calculated by subtracting the current time by its allocation time. If the lifetime is smaller than or within some tolerable range (based on a pre-defined threshold) from the maximal lifetime associated with the corresponding object group, the maximal lifetime remains unchanged, and its *stableTime* is incremented by the elapsed CPU processing time from the last update. Otherwise, the maximal life time is updated to be this object's lifetime and the *stableTime* is reset to zero. Finally, other information, such as the number of current live objects and the total memory space currently occupied by this memory object group, is also updated.

This step is implemented by wrapping the memory allocation/deallocation functions such as *malloc(), calloc(), realloc(), free()*, etc. For programs that use their own memory allocators, we wrap their allocation and free functions.

1530-0897/05 $20.00 © 2005 **IEEE**
IEEE COMPUTER SOCIETY

3.2.2 *Step2: Outlier Detection*

The detection techniques are different for different types of memory leak. For each memory object group, it first checks whether this group has ever called deallocation before. If so, it follows the detection procedure for SLeaks (sometimes-leak). Otherwise, it continues the process of ALeak (always-leak) detection.

To detect ALeaks, SafeMem monitors the memory usage behavior. It first checks whether the number of live objects of each object group exceeds some given threshold. If so, it then checks whether the memory usage by this group is continuously growing. This is done by checking the last allocation time associated with this group. If the last allocation time is long time ago (compared to the current time), the memory usage is not dynamically growing. This is unlikely to be memory leaks. Instead, it might be the case that the program allocates many objects at initialization time and these objects are used throughout the entire execution. However, if the last allocation time is very recent, it indicates that the memory usage is still growing. Therefore, this group of memory objects are leak suspects, which should be monitored using ECC protection for false positive pruning.

To detect SLeaks, SafeMem monitors the lifetime of each live object. An object is singled out as a suspect to be monitored using ECC protection if two conditions hold: (1) this object has been alive for more than two times its expected maximal lifetime, and (2) the maximal lifetime for the corresponding object group has been relatively stable for a period of time (longer than a given threshold). If condition 2 is not true, no outliers will be singled out because the detection confidence is very low in such cases. Because all live memory objects of the same group are linked in the order of their allocation time, SafeMem only needs to check the top few oldest memory objects' lifetimes to detect potential SLeaks.

The detection process is triggered after a warm-up period, and is periodically performed only at memory allocation/deallocation time. More specifically, at each memory allocation/deallocation, if the elapsed time from the last check is greater than a pre-defined parameter, called the *checking-period*, the detection process is performed. Therefore, this step has a very small overhead.

It is safe to perform the detection process only at memory allocation/deallocation time. If the program has not performed any allocation/deallocation for a long time, there is no need to trigger the detection process because the memory usage is not actively growing. Therefore, even if some memory objects have already been leaked, it will not cause the program to crash since the memory usage has stopped growing. As mentioned before, our study focuses on detecting continuous memory leaks that can affect system reliability and availability.

3.2.3 *Step3: False Positive Pruning Using ECC Protection*

When an object is marked as a suspect during Step 2, it is monitored using ECC protection to prune false positives from real leaks. This is based on the observation that if a suspect is accessed again, it is unlikely to be a memory leak. If it has never been accessed for a threshold of time, it is reported as a memory leak.

The pruning procedure works as follows. Each suspect is monitored by calling *WatchMemory*. The first access to this suspect will trigger the ECC protection handler which then removes this object from the suspect list and turns off the ECC monitoring for this object. If this suspect is an SLeak suspect, this object's allocation time is reset to the current time to catch possible future leaks (an object can become a suspect again if it continues to live longer than the expected maximal lifetime). The maximal lifetime associated with this object group is then updated to be the current living time of this suspect to avoid other similar false positives.

The pruning process does not impose significant overhead since it is only performed on rare suspects. In addition, only the first access to a suspect needs to pay the extra overhead of triggering and executing the ECC fault handler.

4 Detecting Memory Corruptions

Memory corruption can be caused by many reasons, among which buffer overflow and accesses to freed memory are two of the most common. Buffer overflow is a particularly important type of memory corruption because it is often exploited by viruses to attach and execute malicious code. Therefore, SafeMem focuses on detecting buffer overflows and accesses to freed memory, both of which are also the major types of bugs detected by Purify.

To detect buffer overflow, SafeMem pads the two ends of each buffer and then uses ECC protection to guard these paddings; any accesses to the padding are reported as buffer overflow bugs. The current implementation of SafeMem uses a cache line as the padding unit. It could easily use longer paddings, but our experiments on applications with buffer overflow bugs show that the current setting is good enough. To reduce false sharing, each memory buffer is cache line aligned. When a buffer is deallocated, the ECC monitoring of its paddings is disabled.

To detect accesses to freed memory, SafeMem uses ECC protection to watch all freed memory buffers. An access to such a buffer will trigger the ECC fault handler which reports this access as a bug. When a freed memory buffer is reallocated, ECC monitoring for this buffer will be disabled. Similar to buffer overflow detection, each memory buffer and its size need to be cache line-aligned to avoid false sharing.

IEEE COMPUTER SOCIETY

The overhead to detect both types of memory corruption is relatively small because it only needs an extra system call at the memory allocation/deallocation time. Since most programs do not have very frequent allocation/deallocation, the overhead imposed by SafeMem is small, as shown in our experimental results (See Section 6).

ECC protection can also be used to detect other types of bugs, even though the current implementation of SafeMem does not support them yet. For example, accesses to uninitialized objects could also be detected using ECC protection. After a memory buffer is allocated, it can be protected using ECC protection. The first write to this buffer would disable the ECC protection, but the first read would be detected and reported as a bug.

5 Methodology

5.1 Platform

Our experiments are conducted on a real system with a 2.4 GHz Pentium processor, an ECC memory controller with the Intel E7500 chipset [18], and 1 GByte of memory. Our operating system extensions (the three new system calls) are added into Linux kernel 2.4.20. SafeMem is implemented as a shared library and can be dynamically preloaded in advance to avoid recompilation of the tested programs (unless the programs use their own memory allocators, in which case we need to do some simple changes to intercept their memory allocation/deallocation calls).

In our evaluation, we compare the time overhead of SafeMem to Purify [15], a state-of-the-art dynamic bug detection tool. Purify can detect memory corruption and memory leak bugs. More specifically, in order to find memory-access errors, Purify maintains two bits for each byte of memory to track its status: allocated or freed, and initialized or uninitialized. Purify checks each memory operation against its status and reports illegal accesses. As for memory leaks, at some point during program execution or when the tested program exits, Purify applies an algorithm similar to the conventional mark-and-sweep algorithm [15], which utilizes conservative pointer tracking to scan the whole heap. Performing such an expensive operation adds large overhead and also significantly perturbs the program's response time, especially for server programs. Therefore, these tools are always used for in-house debugging instead of during production runs.

We evaluate seven different real-world, buggy applications shown on the Table 1, from complicated network server daemons, such as squid and proftpd, to simple common utilities, e.g., gzip. We can divide these tested applications into two groups: one containing memory leaks, and the other containing memory corruption bugs.

Based on these applications, we have conducted two sets of experiments. The first set evaluates the functionality of SafeMem in detecting bugs, and the second set compares the overhead of SafeMem to Purify's using bug-free runs of the tested applications (with normal inputs). In addition, we also evaluate the benefits of ECC protection in reducing memory waste and pruning false positives.

Bugs	Application	LOC	Description
Memory Leak	ypserv1	11,200	a NIS server
	proftpd	68,700	a ftp server
	squid1	95,000	a Web proxy cache server
	ypserv2	9,700	a NIS server
Memory Corruption	gzip	8,900	a compression utility
	tar	34,000	an archiving utility
	squid2	93,000	a Web proxy cache server

Table 1: Tested Applications (LOC means lines of code. squid1 and squid2 are different versions of squid, but one contains memory leaks and the other contains a memory corruption bug. Similarly, ypserv1 and ypserv2 are different versions of ypserv, but one contains ALeaks and the other contains SLeaks).

Even though several previous studies [23] have directly compared their tools with Purify for detecting only one type of bug, memory corruption, we do note that Purify can check for other types of bugs, such as accesses to uninitialized variables, which are not detected by SafeMem. Unfortunately, the current version of Purify does not provide options to allow us to disable these checks to make the comparison fair. However, based on our experience and understanding of Purify's techniques, disabling these checks would not reduce its overhead significantly. After all, Purify needs to monitor every memory access no matter whether it is for detecting memory corruption or for detecting accesses to uninitialized variables. Moreover, this does not have much impact on the interpretation of our results since SafeMem has a substantial overhead reduction (by orders of magnitudes) over Purify.

6 Results

6.1 Microbenchmark Results

First we conduct some microbenchmarks to measure the cost of the ECC monitoring system calls. Table 2 shows the cost for the *WatchMemory()* and *DisableWatchMemory()* system calls on our machine. The costs for these two calls are relative cheap (less than 2 microseconds), compa-

	Calls	Time(microseconds)
ECC Protection	WatchMemory	2.0
	DisableWatchMemory	1.5
Page Protection	mprotect	1.02

Table 2: Time for the ECC system calls

IEEE COMPUTER SOCIETY

Bugs	Application	Bug Detected?	SafeMem Overhead(%) of Detecting			Purify Overhead (%)	Reduction by SafeMem
			Only ML	Only MC	ML + MC		
Memory Leak (ML)	ypserv1	YES	1.0	4.2	6.0	941	157X
	proftpd	YES	0.9	2.6	3.6	2093	581X
	squid1	YES	5.6	7.8	13.7	1782	130X
	ypserv2	YES	0.7	10.5	11.5	1308	114X
Memory Corruption (MC)	gzip	YES	0.3	2.2	3.0	4979	1660X
	tar	YES	0.7	1.0	1.6	475	297X
	squid2	YES	6.1	8.1	14.4	1720	119X

Table 3: Time overhead (%) comparison between SafeMem and Purify

rable to the page protection call *mprotect()* provided by the standard Linux system. Ours are slightly higher than *mprotect* because our calls need to pin (unpin) the page in the virtual memory system.

6.2 Overall Results

Table 3 shows the overall results of SafeMem with seven buggy applications. First, SafeMem can detect all the tested bugs (both memory leaks and memory corruption). This shows that SafeMem is effective in achieving its expected functionality.

We also compare SafeMem's overhead with Purify's. For fair comparison, *SafeMem enables both memory leak detection and memory corruption detection for all experiments, even though each application has only one type of bug.* To avoid disturbance by the bugs, we use normal inputs when we measure overheads so the bugs do not occur and program can run correctly to completion.

As shown in Table 3 (column "ML+MC"), SafeMem adds only 1.6%-14.4% overhead for all tested applications, a factor of 114-1660 times smaller than Purify's overhead (4.8X - 49.8X). For example, for gzip SafeMem adds only 3.0% overhead, whereas Purify slows down this application by a factor of 49.8. This is because SafeMem does not need to monitor each memory access. Instead, it relies on ECC protection and intelligent memory usage behavior analysis to detect memory corruption and memory leaks. In contrast, Purify needs to intercept every memory access in order to detect memory corruption, and needs to do a mark-and-sweep over the entire memory space in order to detect memory leaks. Our small overhead indicates that SafeMem can be used to detect memory leaks and memory corruption *during production runs*.

We further measure SafeMem's overhead for detecting only memory leaks and detecting only memory corruption, respectively. The memory leak detection overhead comes mainly from the information collection and analysis, whereas the memory corruption overhead comes mainly from the ECC monitoring and unmonitoring. Table 3 also shows that overhead caused by memory corruption detection is more than that caused by memory leak detection. This is because memory corruption detection needs to enable ECC monitoring at each buffer allocation and disable ECC monitoring at each deallocation. Memory leak detection, however, only enables monitoring for the suspected memory objects, which usually is many fewer than the total number of allocated memory objects.

6.3 Benefits of ECC Protection

Table 4 shows the benefit of ECC protection over page protection in reducing memory waste for padding and alignment. As shown on this table, ECC-protection adds only 0.084%-334% of total memory overhead (not necessarily used at the same time) for the tested applications, whereas page-protection has 6.06%-231.78X of memory space overhead! In other words, ECC-protection can reduce the memory waste of page-protection by a factor of 64-74! This shows that ECC protection is a better mechanism to use for detecting memory leaks and memory corruption.

Bugs	Application	Memory Overhead(%)		Reduction by ECC
		ECC-	Page-Protection	
Memory Leak	ypserv1	57	3900	68X
	proftpd	35	2357	67X
	squid1	26.4	1950	74X
	ypserv2	3.6	233	64X
Memory Corruption	gzip	0.084	6.06	72X
	tar	334	23178	69X
	squid2	28.7	2120	73X

Table 4: Comparison of space overhead (%) of ECC-protection based approach vs. page-protection based approach. The overhead is calculated over each applications' actual memory usage throughout the whole execution.

6.4 Effects of ECC-Protection in False Pruning for Memory Leaks

Table 5 reports the effects of ECC-protection in false pruning for memory leaks. The results show that this pruning mechanism is very effective: it is able to reduce the number of false positives from 2-13 to 0-1. For example, for squid1, without this pruning scheme, SafeMem would have introduced 13 false positives instead of 1 false positive, which is much harder for programmers. SafeMem does not have any false positives in memory corruption detection because any accesses to padding areas or freed memory buffers are true memory corruption.

1530-0897/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

Application	False Positives	
	Before Pruning	After Pruning
ypserv1	7	0
proftpd	9	0
squid1	13	1
ypserv2	2	0

Table 5: False memory leaks reported before and after using ECC-protection (No false positives for memory corruption detection by SafeMem)

7 Related Work

7.1 Memory Leak Detection

Much research has been conducted on addressing the memory leak problem. Garbage collection [30, 4] is a commonly-used approach to avoid memory leaks in programs. However, it works safely only for type-safe languages such as Java. Therefore, for server programs that are typically written in C/C++, this method is seldom used. In addition, garbage collection also incurs high overhead to perform mark-sweep operations. Such high overhead can significantly perturb the response time of server programs. Another method relies on language support. For example, linear types allow only one reference to each dynamically allocated object [29]. Once a linear-typed variable is read, its content is nullified. Even though memory management is simplified with this method, it is difficult to use such semantics to write programs.

So far two approaches have been proposed to detect memory leaks for server programs written in C/C++. The first approach uses static program analysis to catch potential memory leaks without executing the program. Examples of static tools include PREfix [5], METAL [14] and Clouseau [16]. Clouseau is a recently proposed static checker that has improved previous static tools by avoiding global pointer aliasing analysis using an object ownership model. Promising results have been shown for using this tool to detect many memory leaks in C/C++ programs. However, similar to other static checkers, this tool has a lot of false positives since it does not have the accurate information available only during execution. In addition, as the authors of Clouseau have acknowledged, this tool cannot handle type casting, pointer arithmetic, arrays of pointers, address of a pointer member field in a class or structure, concurrent execution and exception handling [16]. These limitations significantly restrict its usage in detecting memory leaks for large server programs. In contrast, SafeMem does not have these restrictions since it is based on memory usage behavior analysis instead of program static analysis.

The other approach detects memory leaks dynamically at run-time. Examples of dynamic tools include the state-of-the-art Purify [15] and Valgrind [26]. These tools monitor every dynamically allocated memory object, and report leaked memory by mark-sweeping the virtual memory for unreferenced objects. While these tools do not suffer from the same limitations as static tools, mark-sweeping the entire virtual address space can add significant overhead, especially for server programs that usually have large address spaces for buffering or caching. During a mark-sweeping operation, the execution of the program needs to pause to avoid inconsistency, which makes the service unavailable during the entire mark-sweeping operation. Therefore, these tools are always used for in-house debugging instead of during production runs. Our experiments have shown that SafeMem has significantly less overhead than Purify.

7.2 Memory Corruption

Many tools have been proposed for detecting memory corruption. They can be classified into static tools and dynamic tools. In this section, we briefly discuss those that are not described in the earlier sections.

Static tools check for memory corruption statically using program analysis. For example, LCLint [11] is an annotation-assisted lightweight static checking tool. It has been extended by Evans and Larochelle [12]. They exploit semantic comments that are added to source code and standard libraries to detect likely buffer overflow. CSSV, proposed by Sagiv et al. in [10], statically detects unsafe string operations in C programs with the aid of procedure summaries. Though more accurate, writing procedure summaries imposes an extra burden on the programmer.

Dynamic tools check for memory corruption at run time. Examples of dynamic tools include Purify [15], CCured [23, 7], SafeC [1], Jones and Kelly's tool [19], and StackGuard [9]. StackGuard focuses only on stack smashing bugs, ignoring other types of memory corruption.

Purify instruments the object code at link time and does not require source code changes. However, to detect memory corruption bugs, such as buffer overflow or accesses to freed memory, Purify needs to intercept every memory access, which incurs very high overhead, up to a factor of 50.

Jones and Kelly's tool [19], PointGuard [8], SafeC [1] and CRED [24] can detect buffer overflows by dynamically checking each pointer dereference. However, these tools require pointer-object associations in order to find whether a pointer is out-of-bounds. These tools fail when such associations are not available (because of fine-grained pointer manipulation through various type-casting) or when the bug does not violate pointer-type/object association (such as a wrong pointer assignment bug caused by copy-paste). Our tool does not have such limitations, since SafeMem does not require any pointer-object association. It simply detects invalid accesses to monitored areas, no matter what variable name such an access uses.

CCured [23, 7] is a hybrid static and dynamic bug detection tool. It first attempts to enforce a strong type system

IEEE COMPUTER SOCIETY

in C programs via static analysis. Portions of the program that cannot be guaranteed by the CCured type system are instrumented with run-time checks to monitor the safety of executions. Cyclone [13] is very similar. It changes the pointer representation to detect pointer dereference error. In addition to the same limitations as SafeC and CRED, CCured and Cyclone require non-trivial changes to applications' source code to conform to their C standard. In contrast, SafeMem requires little change to programs. In addition, SafeMem can also detects memory leaks.

iWatcher [32] is another related work. It can also monitor accesses to watched locations. Even though it imposes less overhead than ECC protection, it requires extension to the existing microprocessor. In contrast, SafeMem does not require any extension and can work in existing systems with ECC memory support.

7.3 Other Related Work

Compiler lifetime analysis has been used in the garbage collection [3, 17] as an optimization to shift some of the runtime overhead to compile-time. Dynamic lifetime analysis has also been used in the garbage collection [21, 27] and dynamic memory allocation [2]. Based on profiling information, previous work divides memory objects into two groups: short-lived and long-lived, and applies faster memory allocation or garbage collection methods to short-lived objects. Our method for memory leak detection is based on dynamic lifetime analysis. Instead of improving performance as done by previous work, our work focuses on detecting memory leaks, which requires more accurate lifetime information and more intelligent lifetime analysis.

As we mentioned earlier in Section 2, our work is also related to previous research on shared virtual memory systems such as IVY [20], especially those on fine-grained distributed shared memory systems (DSMs) such as Blizzard [25]. Those works use page-protection or ECC protection to implementation cache-coherence operations, whereas our work uses it for software debugging. Therefore, the design trade-offs are different.

8 Conclusions

This paper presents an approach called SafeMem that makes a novel use of ECC memory for detecting memory leaks and memory corruption, two major forms of software bugs that contribute significantly toward software vulnerabilities. Our approach does not require any new hardware extensions and can work with existing systems with ECC memory, which is commonly used in modern systems. Moreover, we also present a new method that uses intelligent memory usage behavior analysis to detect memory leaks.

We have evaluated SafeMem using seven real-world buggy applications. Our results show that SafeMem can detect all tested bugs with only 1.6%-14.4% overhead, 2-3 orders of magnitude smaller than the commonly used commercial tool, Purify. These results indicate that SafeMem can be used for on-the-fly detection of memory leaks and memory corruption during production runs. Moreover, our results also show that ECC protection can reduce the amount of wasted memory by a factor of 64-74 compared to page protection. Finally, ECC protection is also very effective in pruning false positives for memory leak detection.

We plan to extend our work in several dimensions in the future. First, we have evaluated SafeMem with a limited number (only seven) of applications since it is very difficult to find real-world applications that contain well-documented bugs (e.g. what inputs to use in order to generate the bug). Second, we plan to compare SafeMem with other tools. Unfortunately, most existing tools are not publicly available, and some available tools are either unable to support C/C++ programs or require significant modification to applications to conform to their standard. Third, we plan to investigate how to use ECC memory for other software debugging problems.

9 Acknowledgments

The authors would like to thank the anonymous reviewers for their invaluable feedback. We also thank Sanjeev Kumar (Intel) for useful information on ECC chipsets. We appreciate useful discussions with Wei Liu and the OPERA group. This research is supported by the IBM Faculty Award, NSF CNS-0347854 (Career Award), NSF CCR-0305854 grant and NSF CCR-0325603 grant. Our experiments were conducted on equipments provided through the IBM SUR grant.

References

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)*, pages 290–301, Jun 1994.

[2] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation(PLDI)*, pages 187–196, Jun 1993.

[3] J. M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, 1977.

[4] H.-J. Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI)*, pages 197–206, Jun 1993.

1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

[5] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software : Practice and Experience*, 30(7):775–802, 2000.

[6] CERT/CC. Advisories. http://www.cert.org/advisories/.

[7] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 232–244, Jun 2003.

[8] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Aug 2003.

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, Jan 1998.

[10] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 155–167, Jun 2003.

[11] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 87–96, Dec 1994.

[12] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[13] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, Jun 2002.

[14] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 69–82, Jun 2002.

[15] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 125–136, Dec 1992.

[16] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 168–181, Jun 2003.

[17] J. E. Hicks, Jr. *Compiler-Directed Storage Reclamation Using Object Lifetime Analysis*. PhD thesis, Laboratory for Computer Science, MIT, 1992. Available as Technical Report MIT/LCS/TR-555.

[18] Intel. Intel e7500 chipset datasheet. http://www.intel.com/design/chipsets/e7500/datashts/290730.htm.

[19] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, pages 13–26, May 1997.

[20] K. Li. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing (ICPP)*, volume II Software, pages 94–101, Aug 1988.

[21] H. Lieberman and C. E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[22] Microsoft. The common language runtime (CLR). http://msdn.microsoft.com/netframework/programming/clr/default.aspx.

[23] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 128–139, Jan 2002.

[24] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *the 11th Annual Network and Distributed System Security Symposium (NDSS)*, pages 159–169, Feb 2004.

[25] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 297–306, Oct 1994.

[26] J. Seward, N. Nethercote, and J. Fitzhardinge. Valgrind, an open-source memory debugger for x86-gnu/linux. http://valgrind.kde.org/.

[27] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SESPSDE)*, pages 157–167, Apr 1984.

[28] US-CERT. US-CERT vulnerability notes database. http://www.kb.cert.org/vuls.

[29] P. Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359, Apr 1990.

[30] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management (IWMM)*, pages 1–42, Sep 1992.

[31] 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 (ASPLOS)*, pages 304–316, Oct 2002.

[32] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architecture support for software debugging. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, pages 224–237, Jun 2004.

1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY