Tradeoffs in Fine-Grained Heap Memory Protection [*]

Jianli Shen, Guru Venkataramani, Milos Prvulovic
College of Computing
Georgia Institute of Technology
{jianli, guru, milos}@cc.gatech.edu

ABSTRACT

Different uses of memory protection schemes have different needs in terms of granularity. For example, heap security can benefit from chunk separation (by using protected "padding" boundaries) and meta-data protection. However, such protection can be done at different granularity(eg. per-word, per-block, or per-page), with different performance, cost and memory overhead tradeoffs for different applications. In this paper, we explore these tradeoffs for the purpose of heap security in order to discover whether the "right" granularity exists and how the granularity of protection affects design decisions.

We evaluate such tradeoffs based on the current heap-security approaches in a single address space operating system.The access control granularities we use are word, 8-byte, 16-byte, 32-byte, and page. We find that none of these schemes is optimal across all applications. In some applications, excessive padding degrades caching performance for coarse-granularity schemes, while in others, large-block permission changes introduce large overheads for finer granularities. To overcome these limitations, we propose a new two-granularity scheme, which uses word- and page-granularity protection to eliminate padding but allow fast page-size permission changes for large memory blocks. On all applications, this new scheme performs as well or better than the best single-granularity scheme. It also performs on par with the more complex Mondrian Memory Protection, which uses a complex trie structure and multiple permissions caching mechanisms to support a hierarchy of protection granularities.

[*]This work was supported, in part, by the National Science Foundation under Grant Numbers CCF-0429802, CCF-0541080 and CAREER award 0447783. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Categories and Subject Descriptors

C.0 [**Hardware/software interfaces**]:

Keywords

Heap Security, Memory Protection, Protection Granularity

1. INTRODUCTION

In recent years, there has been a clear trend toward improving security of computer systems and resilience against attacks. This trend, combined with increasing number of transistors on a chip, has led to a number of proposals for hardware-based and hardware-assisted protection schemes.

Many attacks today exploit some heap-based vulnerabilities, such as buffer overflows [5], or a function pointer [3] overwrites with input data, modification of heap meta-data, etc. As a result, hardware-enforced memory protection of smaller-than-page memory regions (e.g. write-protect function pointer of a heap block metadata) would go a long way toward improving security of computer systems.

Unfortunately, there have been relatively few studies of such fine-grain memory protection, especially when it comes to tradeoffs between hardware and software support. In this paper, we take an approach of selecting one particular (but fairly broad) use of fine-grain protection, and then examining how to achieve such protection with different levels of hardware support. Our scheme individually protects heap chunks from sequential overflows in neighboring chunks, and also prevents corruption of heap meta-data. The levels of hardware support we examine range from using existing page-granularity protection, to word-granularity protection where every memory word in the application's address space has its own set of access permissions.

To implement the desired heap protection using hardware support that has a protection granularity larger than a word, we rely on software padding to block-align memory areas that have different protection. Once we account for all the overheads of checking and manipulating (modifying) permissions, we find that fine-grain schemes degrade performance in applications with large heap objects, because permissions changes require iterating through many small permissions entries. However, coarse-grain schemes degrade performance for applications with small objects, because alignment wastes space and results in increased cache and TLB pressure.

As a result of these considerations, we propose a new two-level scheme that supports word- and page-granularity protections. Word-granularity support eliminates padding and its cache- and TLB-related problems, while page-granularity protection allows quick permission changes for large blocks that "cover" entire pages. We find that this new scheme performs, on each application, as well or better than the best single-granularity scheme for that application, with performance overheads below 2% on all applications and well below 1% on average. We also find that this simple two-granularity scheme slightly outperforms the more complex Mondrian Memory Protection (MMP), which uses a trie structure to keep its protections with a hierarchy of granularities, and uses multiple sophisticated caching mechanisms to quickly look up these protections.

The rest of the paper is organized as follows: Section 2 gives an overview of fine-grained heap protection, Section 3 presents our experimental setup, and Section 4 presents the evaluation of protection characteristic of single-granularity schemes. Based on these findings, Section 5 presents our new simple and effective two-level scheme and compares its performance to other schemes. Section 6 presents related work. Finally, Section 7 presents our conclusions and future work.

2. OVERVIEW OF FINE-GRAIN HEAP PROTECTION

The heap memory area is typically managed by a user-level library, such as the C standard library (e.g. glibc [7]). Data within the heap area is divided into chunks of various sizes. Allocation requests are serviced by finding a suitably-sized free chunk. If there are no chunks of the right size, a larger chunk is split to obtain a new chunk of a needed size. Finally, if there are no more chunks that are large enough, the heap area is expanded using a *brk* system call or a new heap area is created using *mmap*.

To quickly service allocations, free chunks are organized into *free lists*, with each free list containing chunks in a given size range. As a result, most often a free chunk of appropriate size can be found at or near the head of the free list for that chunk size.

A deallocated chunk can simply be added to a free list for that chunk size. However, the heap management library tries to reduce heap fragmentation through *consolidation* of free chunks. When a chunk is freed, the library checks the neighboring chunks, and if one or both neighbors are free they are combined with the newly freed chunk into a larger free chunk.

Meta-data for a heap chunk consists of the chunk's data size and allocation status, and free list pointers for a free chunk. This metadata must be quickly found given a chunk's address, so a typical heap library keeps its meta-data interleaved with data - meta data is kept as a header that precedes the chunk's data region. Given the chunk's data address (e.g. when *free()* is called), its meta-data is found at a constant (negative) offset.

Unfortunately, such meta-data is easy to corrupt. For example, if a heap chunk contains a buffer, writing past the end of the buffer will result in modifying meta-data for the next chunk in the heap memory area. This can result in leaking memory (e.g. by reducing the size of a large free chunk) or crashes (e.g. by changing the status of an allocated chunk to appear free). By leveraging the free list insertion/deletion code, the attacker can even manipulate free list pointers in a way that causes a value of attackers choice to be placed in a memory location of attackers choice [1, 10]. Note that this is a very powerful exploit for attackers, e.g. it allows a function pointer to be modified directly without affecting surrounding data, which defeats protection schemes based on cannary values [4].

Heap data is also vulnerable to attacks, e.g. by a buffer overflow that overwrites data in the next chunk which happens to contain function pointers or other information that can affect control flow.

To protect heap data and its corresponding meta-data against sequential buffer overflows, we add a write-protected block before and after meta-data, as shown in Figure 1. With a buffer overflow reaches one of these protected block before it reaches any meta-data (or data in another chunk).

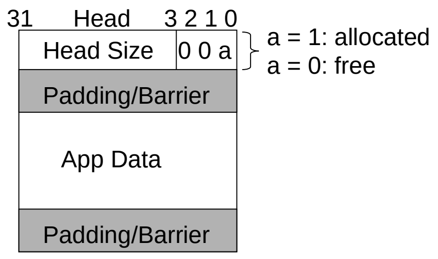

Figure 1: Heap Structure with Interleaved Padding.

However, meta-data of a chunk is usually small (e.g. only 4 bytes in the GNU standard C library on a 32-bit architecture), and some heap chunks also have relatively small data. If the hardware can track protections at the granularity of N bytes, we need two N-byte blocks per chunk and, if N is larger than the word size, additional padding is needed to align data and meta-data blocks to N-byte boundaries.

3. EXPERIMENTAL SETUP

3.1 Protection Information Representation

To study the tradeoffs in fine grained memory protection scheme, we use 2-bit protection entries, with protections as shown in Figure 2. Each protection entry is associated with an N-byte block of memory, where N is the protection granularity. The protection information is kept as a simple array structure in a separate and protected part of the application's address space not accessible to the programmer, and can be stored in memory and cached in L2 on-chip caches. We assume that the OS protects this information through a trusted kernel. Because a protection check is needed every time data is accessed, keeping blocks of protection entries in the data L1 cache would dramatically increase the required L1 bandwidth. To avoid this, we add a separate small L1 protection cache. This cache is only 2KBytes in size, so it is smaller and faster than data and instruction L1 caches. However, since protection entries are much more compact than the corresponding data and code, the protection cache tends to hit more often than existing L1 caches. As a result, a protection cache lookup is very rarely on the critical path of the processor's execution. Finally, the separate L1 protection cache leaves existing highly-optimized L1 caches unchanged, so the overall impact of our fine-grain permissions' checking should be minimal.

Perm Value	Meaning
00	no perm
01	read-only
10	read-write
11	execute-read

Figure 2: 2 Bit Permission Values and Their Meaning.

3.2 Benchmark Applications

We use all C-language applications from the SPEC CPU 2000 [17] benchmark suite except *perlbmk* and *vortex* which use system calls that are currently not supported in our simulator. We modified the standard C malloc library [7] to insert padding and protect/unprotect code for different protection granularities. We compiled all the applications with our new library at the highest (O3) optimization level in GCC. For each application, we use the reference input set in which we fast-forward through the first five billion instructions to skip initialization phases and simulate the next two billion instructions in detail. However, we keep track of the protection information during fast forwarding of instructions to maintain correctness.

3.3 Benchmark Characteristics

To characterize memory allocation behavior in SPEC applications, we show in Figure3 the average allocation request size (in bytes). The figure shows that applications with frequent heap requests have small average allocation sizes (for example, twolf has an average request around 32 bytes, and art request 26 bytes on average). For those applications, the overhead of padding will significantly depend on the protection granularity, both in terms of memory space and in terms of performance.

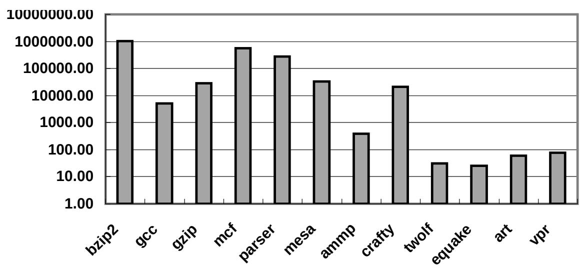

Figure 3: Average Allocation Request Size in Bytes.

3.4 Simulation Environment

We use SESC [11], an open-source execution-driven simulator, to simulate an out-of-order superscalar processor whose architectural parameters are shown in Table 1. The L1 data cache we model is 16KB in size, two-way set associative, dual-ported, with 32-byte blocks. The processor-memory bus is 128 bits wide and operates at 500MHz, and the no-contention round-trip main memory latency is 375 cycles.

The protection cache is 2KBytes in size, 8-way associative and has a 32-byte block size. Its access latency of this cache easily matches the 2-cycle latency of other L1 caches (there is no need for permissions to arrive before the data does).

Architectural Parameters	Specification
ArchBits	32
Clock Frequency	5 GHz
L1 I-Cache	WB 16KB,2way, 32B line
L1 D-Cache	WB 16KB,2way, 32B line
L2 Cache	Unified WB, 2M, 8way, 32B line
L1 latency	2 cycles
L2 latency	10 cycles
State Cache(SL1)	WB 2KB, 8way, 32B line
SL1 latency	2
I/D TLB	128 entries, fully assoc.
TLB miss latency	500 cycles
Memory bandwidth	3.2 GB/sec
Memory latency	375 cycles
Fetch/Decode Width	6/6 per cycles
Issue/Retire Width	6/6 per cycles
ROB entries	156
load/store entries	24/24
Int/float registers	128/128

Table 1: Parameters of the Simulated A rchitecture.

4. PROTECTION GRANULARITY CHARACTERIZATION

In this section, we assume simple hardware support that maintains protection information at a fixed granularity. In other words, one protection entry is associated with each N-byte block of data, where N is hard-wired and protection entries are kept as an array in memory and cached on-chip as described in Section 3.

The smallest protection granularity (the value of N) in our experiments is 4, which corresponds to per-word protection that needs no padding to align protected blocks with a protection granularity boundary. We consider larger granularities of 8, 16, 32, and 64 bytes, which require some padding, as well as a 4096-byte granularity, which corresponds to per-page protection that can use existing page tables and TLBs.

Table 2 shows the heap memory space overhead due to padding to align our protected blocks to protection granularity boundaries. Note that this overhead does not include the two protected "delimiter" needed to "guard" each meta-data entry. This overhead also does not include actual protection entries, whose overhead can easily be calculated as two bits per N bytes of memory mapped into the application's address space. The overhead of protection entries is 6.25% (2/32) for the per-word protection granularity, and diminishes in linear proportion to the protection granularity.

From the table, we observe that the padding overhead in several applications becomes considerable when the protection granularity is 16 or larger. The applications with low average allocation request sizes namely twolf, equake, art and vpr exhibit this behavior. (Figure 3). With a 64-byte protection granularity, in these applications the space used by padding exceeds the space used for actual heap data, and with page-granularity protections the padding is so prevalent that actual data represents less than 2% of the heap area being used.

The execution time overhead for different protection granularities is shown in Figure 4. We observe that several applications suffer huge overheads with page-granularity protec-

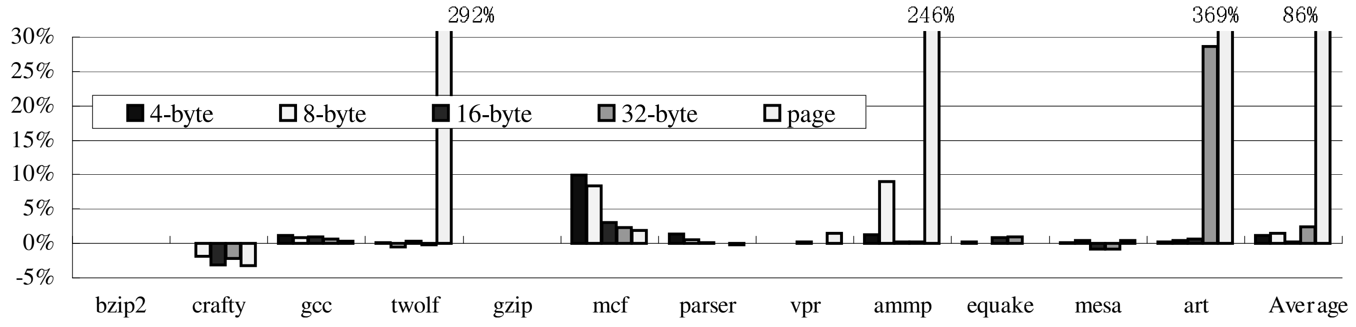

Figure 4: Execution Time Overhead with Different Protection Granularities

bench	*8_byte*	*16_byte*	*32_byte*	*64_byte*	*page*
bzip2	0.00%	0.00%	0.00%	0.00%	0.02%
gcc	0.01%	0.16%	0.53%	1.03%	17.18%
gzip	0.00%	0.00%	0.00%	0.01%	0.26%
mcf	0.00%	0.00%	0.00%	0.00%	0.01%
parser	0.00%	0.00%	0.00%	0.00%	0.07%
mesa	0.00%	0.00%	0.00%	0.00%	0.18%
ammp	1.04%	4.74%	11.90%	20.20%	963.17%
crafty	0.02%	0.06%	0.13%	0.28%	18.15%
twolf	3.55%	33.34%	92.84%	221.61%	9666.51%
equake	0.00%	22.25%	61.53%	173.24%	7112.92%
art	0.00%	18.17%	47.33%	134.42%	5575.46%
vpr	0.01%	12.37%	61.39%	159.51%	6226.98%

Table 2: Memory Space Overhead of Padding for Different Protection Granularities

tion. This performance overhead is mostly caused by TLB contention and cache conflicts. Note that we place a protected word between data of one chunk and the meta-data of the next chunk, and we also place a protected word between a chunk's meta-data and its data. With page-granularity protection each of these protected words must be on a page by itself. As a result, meta-data of each chunk is on a page by itself (”sandwiched” by protected pages), and data for a chunk also occupies one or more entire pages. As a result, each actively used heap block uses at least one TLB entry, and memory management operations access a large number of pages (one for each accessed meta-data entry). This is in contrast to “normal” heap access patterns, where data and meta-data for many small blocks may fit in a single page. As a result, some applications experience a greatly increased number of TLB misses. Furthermore, since all small heap chunks occupy cache blocks that are at the beginning of their pages, cache blocks that belong to sets whose tags correspond to a beginning of a page experience heavy contention, while blocks in other sets remain under-utilized. This considerably increases the number of cache conflicts and further degrades performance.

For smaller protection granularities, the performance overhead is caused mainly by two factors. First, finer-granularity protection means that an allocation or deallocation of a large block changes a large number of protection entries. A clear example of such behavior is mcf, which has a very large allocation request size. As the protection granularity increases, fewer protection entries need to be un-protected and the overhead is reduced. The second source of overhead is the increasing cache pollution and conflicts due to padding. The clearest example of this is in art, where a transition from 32-byte to 64-byte granularity results in a large overhead increase (from less than 1% to nearly 30%). This is because a typical heap chunk in art fits in a single 32-bit cache line, and 64-byte alignment results in increased demand for even-numbered cache sets (heap data maps there) and under-utilization of odd-numbered cache sets (padding maps there).

Overall, we observe that small-granularity protection is needed to achieve low memory space overheads and to reduce cache and TLB conflicts, but larger granularity is beneficial to reduce the overhead of changing protections for large chunks of memory. As a result, we find that a granularity of 8 bytes is a good balance between padding space overhead (less than 4% in all applications) and performance overhead (less than 9% in all applications). However, we note that the ideal scheme would be one to allow fine-grain protection for applications that have small heap chunks (to reduce padding overhead and contention in caches and TLBs), while also allowing large-granularity protection in applications that have large heap chunks (to reduce overhead of protection updates).

From Table 2 and Figure 4, we see that an ideal scheme would use multiple granularities. In the next section, we describe a simple two-granularity scheme whose complexity and cost is similar to our word-granularity scheme, but with the added advantage of being able to handle large blocks at page-granularity.

5. A NEW TWO-GRANULARITY SCHEME

The multiple-granularity scheme we propose uses only two granularities: word and page. The word granularity allows us to avoid padding, and the page granularity allows us to quickly update protections when an entire page belongs to the same chunk. For the word granularity, we use the same protection storage and caching approach that was described in previous sections. Our page-granularity protection, however, is kept in the page table and cached in TLBs. The page-granularity protection entry also has an extra bit, which indicates whether the page uses page- or word-granularity protection.

To check access permissions for a given address, the hardware first accesses the TLB, handling TLB misses in the same way as existing processors. In addition to access permissions maintained by the OS, the hardware also checks our 3-bit user-level permissions. If the granularity bit indicates that page-granularity protection is used for that page,

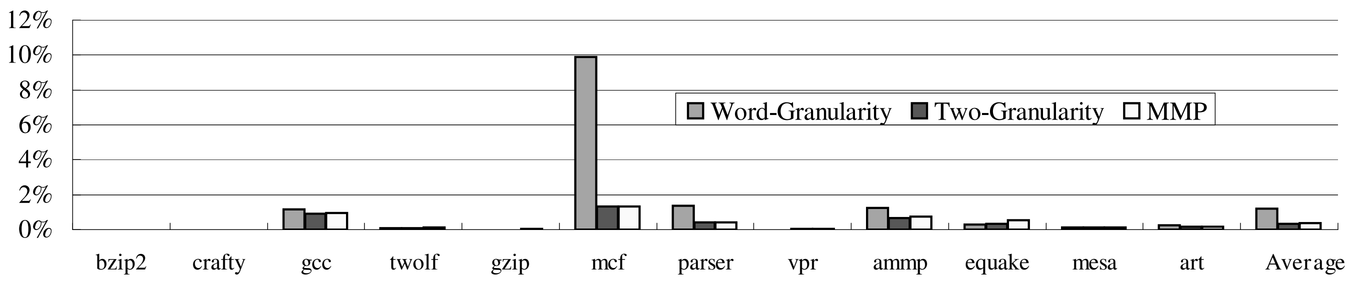

Figure 5: Execution Time Overhead of Our New Two-Granularity Scheme, Compared to Word-Granularity Protection and MMP.

the 2-bit protection entry is used to check whether the current access is allowed. If the word-granularity protection is indicated, the processor searches for the word-granularity protection entry (or entries, in case of a multi-word access) in the L1 protection cache, using it to check whether the current access is allowed. Misses in the L1 protection cache are handled by fetching the block of permissions from the "regular" L2 cache and, if needed, memory.

We note that it would be possible to use a different granularity as the "large" granularity in a two-granularity scheme. However, we choose page granularity becasue it has the advantage of being already supported by page tables and TLBs, so we do not need another protection cache and a second memory area to keep page-granularity protection entries.

The memory space overhead of our two-granularity scheme is similar to the word-granularity scheme (6.25% for per-word protection entries). Note that we still keep word-granularity protection entries allocated, even for pages that use page-granularity protections. This is done to simplify management of protection entries - it allows us to keep per-word protections in a single array structure and avoid complex memory management operations when switching a page of data between protection granularities.

The procedure that sets permissions for a region of memory iterates through that region, and checks whether the next word in the iteration is page-aligned and whether the region "covers" that whole page. If both conditions are satisfied, we set the granularity bit to indicate page-granularity protections, set the page-granularity protection entry, and move on to consider the start of the next page. If the word we are considering is not page-aligned, or the remaining size of the region does not reach the end of the page, we must use word-granularity protections. To do that, we first check the granularity bit in the TLB. If it indicates that word-granularity protection is used for the page, we simply update the word-granularity protection entry and move on to the next word. If page-granularity protection is indicated, we take the page-granularity protection entry and copy it into all word-granularity protection entries for that page. Then we change the granularity bit for the page to indicate word-granularity protection, and finally we update the word-granularity protection entry and move on to the next word. The entire permissions-setting procedure is done in software, and its execution is accounted for in our experimental results.

Figure 5 compares the performance of our two-granularity scheme to the word-granularity scheme. We observe that the two-granularity scheme has overheads below 2% for all applications, in contrast to the single-granularity per-word scheme which has a 10% overhead in mcf.

We also compare our two-level scheme to the more complex Mondrian Memory Protection (MMP) [18]. Mondrian uses an entire hierarchy or protection granularities, and organizes its protection information in a trie data structure. A lowest-level (leaf) node of the trie contains per-word protection entires. Higher-level nodes correspond to increasing protection granularities, and contain either pointers to lower-level nodes or protection entries at the appropriate granularity. Because protection entries at different levels of the trie can correspond to several different granularities, they are cached using a ternary content-addressable-memory (TCAM) protection cache, and also using a set of "sidecar" registers, as described in [18]. We note that the hardware to look up protections in the MMP's trie structure is considerably more expensive than for our simple array-based lookups. MMP's trie structure updates are also considerably more complex than in our scheme. In particular, changes in protection granularity require trie node insertions or deletions. In our experiments, we assume 32-bit virtual addresses, which results in a 3-level MMP trie. With 64-bit addresses, the MMP trie would have 7 levels, causing additional complexity and overheads.

Despite the simplicity of our new two-granularity scheme, its performance is similar to that of MMP. In fact, in most applications our scheme outperforms MMP, although we fully account for the overheads of our scheme and neglect some overheads for MMP.

6. RELATED WORK

Buffer overflow bugs and vulnerabilities have been studied extensively [3, 4, 5], and heap buffer overflows and metadata corruption have been exploited in real attacks [12, 13, 14]. Several software tools for detection of such bugs and vulnerabilities have been developed [8, 15], and more recently hardware support for detection to these and other bugs and vulnerabilities has been proposed [2, 16, 18, 19, 20]. Koldinger et al. [9] discuss architectural implications of single address space operating systems, specifically the interaction between the memory system architecture and the operating system's use of addressing and protection. MIT J-Machine Multicomputer [6] introduces low overhead mechanisms for protection against message corruption, interception and starvation.

Most related to our work is Mondrian Memory Protection (MMP) [18], which uses a trie-based hierarchical memory protection structure. In MMP, the leaf-level protections entries protect memory at a word granularity (one protection entry per 4-byte word), and levels closer to the root of the trie offer protection for increasingly large granularities. This hierarchical organization allows very fine-grained protection where it is needed, while still using large-granularity protection where possible. Unfortunately, the scheme is relatively complex to implement: it requires hardware support to traverse the trie and find the granularity (trie level) that contains the protection entry for a given address, which can result in large overheads. To avoid such overheads, the scheme uses two different caching mechanisms, one of which relies on TCAMs. Furthermore, protection changes are relatively complex and can result in trie node insertions and deletions, which is difficult to support in hardware and, if changes are frequent, can cause noticeable slowdowns if done in software.

In contrast to MMP, our proposed two-level protection scheme relies on existing page tables for its page-granularity protections, and uses a simple array-based structure for its word-granularity protections. As a result, simple indexing can be used to look up our protection entries, and we only add a small and simple L1 protection cache to speed up word-granularity protection lookups (existing TLBs are leveraged to handle page-granularity lookups). This simplicity of our protection structure also allows simpler and faster protection updates, which only need to consider two levels of protections and simple indexing to access protection entries.

A potential disadvantage of our new 2-level scheme is that it can have a larger memory space overhead than MMP when large-granularity protection can be used. Fortunately, page-granularity protections already represent a very minor memory space overhead, and little is gained by using even larger protection granularities. Similarly, our 2-level protection scheme had a potential performance disadvantage because larger-granularity protection information can be cached on-chip more effectively. However, little is gained if protection information is found on-chip but the page table entry is not: a TLB miss must still be serviced, and when the page table entry is brought into the TLB our page-granularity protection is brought with it.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we examine the tradeoffs involved in choosing the granularity of fine-grain memory protection. We find that existing page-granularity protection, when used for fine-grain protection, wastes huge amounts of memory due to padding and results in very poor performance. We also find that a simple word-granularity scheme performs poorly when used to protect large chunks of memory. These observations lead us to propose a new two-granularity scheme that dynamically selects between word- and page-granularity protection for each page. The new scheme is similar in complexity to a single-granularity per-word protection, and leverages existing TLBs and page tables for its page-granularity protection. This two-granularity scheme achieves similar performance but is significantly less complex than MMP, a previously proposed multiple-granularity scheme. Overall, we believe that our new two-granularity scheme is a good tradeoff between performance overhead, memory space overhead, and hardware complexity.

Our future work includes additional uses for fine-grain protection, such as protection for return addresses, function pointers, and read-only data. We also plan to extend our evaluation to include additional benchmarks, especially object-oriented ones that may have additional overheads due to frequent protection changes.

8. REFERENCES

[1] Anonymous. Once upon a free(). *Phrack Magazine*, 57(9), 2001.

[2] M. L. Corliss, E. C. Lewis, and A. Roth. Dise: A programmable macro engine for customizing applications. In *ISCA '03: 30th Intl. Symp. on Computer Architecture*, pages 362–373, New York, NY, USA, 2003. ACM Press.

[3] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities. *in Proc. of the 12th USENIX Security Symp.*, pages 91–104, 2003.

[4] 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 Proc. of the 7th USENIX Security Symp.*, pages 63–78, 1998.

[5] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conf. & Exposition – Volume 2*, pages 119–129, 2000.

[6] W. J. Dally and et al. The j-machine: a fine-grain concurrent computer. In *G. X. Ritter (ed.), Information Processing 89*, North Holland, 1989. Elsevier Science Publishers B.V.

[7] Doug Lea. A Memory Allocator. *http://gee.cs.oswego.edu/dl/html/ malloc.html*, 2000.

[8] IBM Corporation. IBM Rational Purify. *http://www.ibm.com/software/awdtools/purify/*, 2005.

[9] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architectural support for single address space operating systems. In *5th Intl. Conf. on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27, pages 175–186, New York, NY, 1992. ACM Press.

[10] F. Perriot and P. Szor. An Analysis of the Slapper Worm Exploit. *http://securityresponse.symantec.com/avcenter/reference/ analysis.slapper.worm.pdf*, 2003.

[11] J. Renau et al. SESC. *http://sesc.sourceforge.net*, 2006.

[12] Security Focus. Wu-Ftpd File Globbing Heap Corruption Vulnerability. *http://www.securityfocus.com/bid/3581*, 2002.

[13] Security Focus. CVS Directory Request Double Free Heap Corruption Vulnerability. *http://www.securityfocus.com/bid/6650*, 2003.

[14] Security Focus. Sudo Password Prompt Heap Overflow Vulnerability. *http://www.securityfocus.com/bid/4593*, 2003.

[15] J. Seward. Valgrind, An Open-Source Memory Debugger for x86-GNU/Linux. *http://valgrind.kde.org/*, 2004.

[16] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. HeapMon: a Low Overhead, Automatic, and Programmable Memory Bug Detector. In *IBM T.J. Watson Conf. on Interaction between Architecture, Circuits, and Compilers*, 2004.

[17] Standard Performance Evaluation Corporation. SPEC Benchmarks. *http://www.spec.org*, 2000.

[18] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *ASPLOS-X: 10th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, New York, NY, USA, 2002. ACM Press.

[19] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torellas. AccMon: Automatically Detecting Memory-related Bugs via Program Counter-Based Invariants. In *Proc. of the 37th Intl. Symp. on MicroArchitecture)*, 2004.

[20] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proc. of the 31st Intl. Symp. on Computer Architecture*, 2004.