Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software

Chongkyung Kil[*] Jinsuk Jun[*] Christopher Bookholt[*] Jun Xu[†] Peng Ning[*]

Department of Computer Science[*]
North Carolina State University
{ckil, jjun2, cgbookho, pning}@ncsu.edu

Google, Inc.[†]
jxu3@ncsu.edu

Abstract

Address space randomization is an emerging and promising method for stopping a broad range of memory corruption attacks. By randomly shifting critical memory regions at process initialization time, address space randomization converts an otherwise successful malicious attack into a benign process crash. However, existing approaches either introduce insufficient randomness, or require source code modification. While insufficient randomness allows successful brute-force attacks, as shown in recent studies, the required source code modification prevents this effective method from being used for commodity software, which is the major source of exploited vulnerabilities on the Internet. We propose Address Space Layout Permutation (ASLP) that introduces high degree of randomness (or high entropy) with minimal performance overhead. Essential to ASLP is a novel binary rewriting tool that can place the static code and data segments of a compiled executable to a randomly specified location and performs fine-grained permutation of procedure bodies in the code segment as well as static data objects in the data segment. We have also modified the Linux operating system kernel to permute stack, heap, and memory mapped regions. Together, ASLP completely permutes memory regions in an application. Our security and performance evaluation shows minimal performance overhead with orders of magnitude improvement in randomness (e.g., up to 29 bits of randomness on a 32-bit architecture).

1 Introduction

Memory corruption vulnerability has been the most commonly exploited one among the software vulnerabilities that allow an attacker to take control of computers. Examples of memory corruption attacks include buffer overflows [19], format string exploits [20], and double-free attacks [1]. In an attack exploiting a memory corruption vulnerability (or, a memory corruption attack), an attacker attempts to alter program memory with the goal of causing that program to behave in a malicious way. The result of a successful attack ranges from system instability to execution of arbitrary code. A quick survey of US-CERT Cyber Security Alerts between mid-2005 and 2004 shows that at least 56% of the attacks have a memory corruption component [24].

Memory corruption vulnerabilities are typically caused by the lack of input validation in the C programming language, with which the programmers are offered the freedom to decide when and how to handle inputs. This flexibility often results in improved application performance. However, the number of vulnerabilities caused by failures of input validation indicates that programming errors of this type are easy to make and difficult to fix. *Ad Hoc* methods such as StackGuard [8] only target at specific types of attacks. Static code analyzers can be used to find such bugs at compile time. Due to the inherent difficulties in deeply analyzing C code, these analyzers often make strong assumptions or simplification that lead to a significant number of false positives and false negatives. Methods such as CCured [18] offer a way to guarantee that a program is free from memory corruption vulnerabilities. However, such methods incur significant runtime overhead that hinders production deployment. In addition, most of the aforementioned approaches require access to the source code. This is particularly a problem for commodity software, for which the source code is typically unavailable.

While bug detection and prevention techniques are preferred, continued discoveries of memory corruption vulnerabilities indicate alternatives must be sought. We believe that mitigating this kind of attacks would give attackers significantly fewer ways to exploit their targets, thereby reducing the threat they pose. One promising method is *address space randomization* [26]. It has been observed that most attacks use absolute memory addresses during memory corruption attacks. Address space randomization ran-

domizes the layout of process memory, thereby making the critical memory addresses unpredictable and breaking the hard-coded address assumption. As a result, a memory corruption attack will most likely cause a vulnerable program to crash, rather than allow the attacker to take control of the program.

Several address space randomization techniques have been proposed [2, 3, 22, 23, 26]. Among the existing approaches, PaX Address Space Layout Randomization (ASLR) [23] and address obfuscation [3] are most visible. PaX ASLR randomly relocates the stack, heap, and shared library regions with kernel support, but does not efficiently randomize locations of code and static data segments. Address obfuscation expands the randomization to the static code and data regions by modifying compiler but requires source code access and incurs 11% performance overhead on average. Position-independent executables (PIE) [9] allows a program to run as shared object so the base address of the code and data segment can be relocatable, but it also incurs 14% performance degradation on average (shown in our performance evaluation presented later in the paper). While insufficient randomness allows successful brute-force attacks, as shown in recent studies, the required source code modification and performance degradation prevent these effective methods from being used for commodity software, which is the major source of exploited vulnerabilities on the Internet.

In this paper, we propose address space layout permutation (ASLP) to increase the programs' randomness with minimal performance overhead. ASLP permutes all sections (including code and static data) in the program address space. This is done in two ways. First, we create a novel binary rewriting tool that randomly relocates static code and data segments, randomly re-orders functions within code segment, and data objects within data segment. Our rewriting tool operates directly on compiled program executable, and does not require source code modification. We only need the relocation information from the compile-time linker to perform the randomization rewriting. Such information is produced by all existing C compilers. Second, to randomly permute stack, heap, and memory mapped regions, we modify the Linux kernel. Our kernel changes conserve as much virtual address space as possible to increase randomness. Our binary rewriting tool can be automatically and transparently invoked before a program is launched, while our kernel level support runs without any additional change to the runtime system. To validate the practicality and effectiveness of ASLP, we have evaluated ASLP using security and performance benchmarks. Our security benchmark result shows that ASLP can provide up to 29 bits of randomness on a 32-bit architecture, while the performance benchmark result indicates that ASLP incurs less than 1% overhead. In summary, the major contributions of this paper are as follows:

- ASLP provides probabilistic protection an order of magnitude stronger than previous techniques.
- ASLP randomizes regions throughout the entire user memory space, including static code and data segments. Program transformation is automatically done by our binary rewriting tool without the requirement of source code modification.
- The performance overhead is generally very low (less than 1%). In comparison, existing techniques that are capable of randomly relocating static code and data segments, in particular PIE and Address obfuscation, incur more than 10% overhead on average.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the design and implementation of ASLP. Section 4 presents the evaluation of ASLP. Section 5 describes the limitations of current implementation, and Section 6 concludes the paper.

2 Related Work

The seminal work on program randomization by Forrest et al. illustrated the value of diversity in ecosystems and similar benefits for diverse computing environments [10]. In short, their case for diversity is that if a vulnerability is found in one computer system, the same vulnerability is likely to be effective in all identical systems. By introducing diversity into a population, resistance to vulnerabilities is increased. Address randomization achieves diversity by making the virtual memory layout of every process unique and unpredictable. Existing address space randomization approaches can be divided into two categories: user level and kernel level. User level and kernel level approaches achieve randomization with modification to the user space applications and the kernel, respectively. Both approaches have their advantages and disadvantages, which should be carefully reviewed by the user before the application of the techniques.

User Level Randomization Address obfuscation [2, 3] introduced a mechanism to not only randomly shift the placement of the three critical regions, but also randomly re-order objects in static code and data segments. This approach relies on a source code transformation tool [29] to perform the randomization. It introduced special pointer variables to store actual locations of objects for static code and data randomization. In addition, they add randomly sized pads to stack, heap, and shared library using the initialization code, the wrapper function, and the junk code respectively.

Kernel Level Randomization Kernel level randomization has become a more attractive option since modification of one component provides system wide protection.

Recently, kernel level address randomization techniques are being actively used in major Linux distributions: Red Hat Exec-Shield [16] can be found in Fedora Core [6]; PaX Address Space Layout Randomization(ASLR)[23] can be found in Hardened Gentoo [4]. Both PaX ASLR and Exec-Shield use the same approach by padding random size to the critical memory regions. These techniques, unfortunately, have a number of limitations. First, the pads unnecessarily waste memory space. Note that the only way to increase the program randomness is to increase the size of the pads, thereby wasting more space. Second, they keep the relative order of sections. For instance, code segment always comes first, and data segment follows the code segment. Therefore, once an attacker detects the size of the pad, he/she can easily craft attacks to compromise the system. A recent work [21] has shown that the de-randomization attack can defeat PaX ASLR in an average of 216 seconds. This is a clear indication that we need to improve the randomness in the critical memory regions.

3 Address Space Layout Permutation

ASLP provides both user and kernel level randomizations. For user level randomization, we create a binary rewriting tool that randomly relocates the static data and code segments of an executable before it is loaded into memory for execution. Our tool not only alters the locations of static code and data segments but also changes the orders of functions and data objects within the code and data segments. For kernel level randomization, we modified Linux kernel that permutes three critical memory regions. This section explains the design and implementation of ASLP in detail.

3.1 User Level Address Permutation

In a typical program development scenario, a program's source code is written in a high-level language (e.g., *helloworld.c*) and compiled into object files (e.g., *helloworld.o*) that can be linked with other objects to produce the final executable (e.g., *helloworld*). An object file contains not only code (functions) and data (variables), but also additional bookkeeping information that can be used by the compile-time linker to assemble multiple objects into an executable file. Such information includes relocation records, the type of the object file, and debugging related information. In an object file, an element (function, variable, or bookkeeping information) has its own symbolic name. Symbolic names for static code/data elements in object files are resolved to virtual addresses when the executable is assembled by the compile-time linker. Symbolic names for dynamic elements such as C library functions are usually resolved by the runtime system loader.

The goal of our binary rewriting tool is to randomly relocate the static code and data segments and their elements so that the program will have a different memory layout each time it is loaded for execution. This permutation makes it difficult for various types of attacks: partial overwrite attacks [2], dtors attacks [30], and data forgery attacks. These attacks are based on the assumption that the code and data segments of the target application reside at the same locations on different machines. Partial overwrite attacks change only the least significant part of the return address in the stack, so the attacker can transfer the program control to the existing vulnerable function with malicious arguments. Dtors attacks overflow a buffer (global variable) in the data segment to overwrite function pointers in the dtors section. Dtors section includes function pointers that are used after the main() function exits. When a corrupted function pointer is used, the program control is transferred to the attacker's code. Dtors attacks are possible since the dtors section is a part of the data segment and the attacker knows relative distance between the buffer in the data segment and the function pointer in the dtors section. Recent security report [31] shows that data forgery attacks can overwrite existing global variables in the data segment so the attacker can change the value of security critical data. Note that even if kernel level randomization changes the base addresses of stack, heap, and mmap regions, these kinds of attacks can still be successful, since the locations of the code and data segments are fixed.

User level permutation makes these types of attacks significantly difficult. In user level address permutation, we change the base addresses of the static code and data segments. We also randomly reorder procedure bodies within the code segment and data objects within the data segment. As a result, the virtual addresses of static code elements (e.g., functions) or data elements (static or global data variables) are not fixed anymore. To make these changes, we have to modify all cross references between the objects in the program. Otherwise, the randomized program will have many dangling references that will certainly lead to a program crash.

We have developed the binary rewriting tool that transforms an Executable and Linking Format (ELF) [27] executable file into a new one that has a different layout. Our tool allows users to choose any values between 1 and 700K for different offsets for static code and data randomization. The given values are then multiplied by virtual memory page size (usually 4096). Consequently, users can freely decide the starting addresses of the code and data segments in the entire user memory space. (Currently on a default Linux system, user space has 3GB of virtual address space.) Our tool also allows users to change the order of the code and data segments. The default linker always places the data segment after the code segment. Therefore, an attacker can guess the target program layout once he has found certain offset values and the starting address of the code segment. By changing the order of code and data segments, it is more

difficult to guess correct program layout.

Rewriting ELF executable files and making it run exactly as before is non-trivial. There are several challenges during the binary rewriting process, which result in the following questions:

- What parts of an ELF executable file need rewriting?
- How to find the correct locations of those parts and rewrite them?
- How those parts are connected or affect each other at run time? (How functions and variables are referred at run time?)

The first challenge requires that we understand the ELF executable file format and how the linker and the loader create the program memory layout. Currently, an ELF executable file can have totally 45 sections according to the specification. Each section has its own identifier called *section header* which specifies the content of the section and how to access the elements in the section. We found that a total of 13 sections are related to program memory layout. These sections include information about the symbols, dynamic linking, relocation of code and data objects, and other data that are critical to program execution. Table 1 presents the detailed information about the 13 sections[1]. We also found that both the ELF header and the program header need to be modified since they contain an entry point to start the program, the name of the dynamic loader, instructions on what portions of the file are to be loaded, and the permissions of the sections of memory (for example, code segment is read-only). Such information has to be changed once our tool permutes the static code and data segments. (A detailed explanation of rewriting the headers will be discussed later in this section.)

Section Name	Semantics	Section Type
.got	global offset table	SHT_PROGBITS
.plt	procedure linkage table	SHT_PROGBITS
.got.plt	read-only portion of the global offset table	SHT_PROGBITS
.rodata	read-only data	SHT_PROGBITS
.symtab	symbol table	SHT_SYMTAB
.dynsym	dynamic linking symbol table	SHT_DYNSYM
.dynamic	dynamic linking information	SHT_DYNAMIC
.rel.dyn	relocation information for dynamic linking	SHT_REL
.rel.plt	relocation information for .plt segment	SHT_REL
.rel.init	relocation information for .init segment	SHT_REL
.rel.text	relocation information for .text segment	SHT_REL
.rel.data	relocation information for .data segment	SHT_REL
.rel.fini	relocation information for .fini segment	SHT_REL

Table 1. ELF sections to change

The next challenge lies in the method to find out correct locations of the elements in an ELF executable file. We acquire this information by looking up the symbol table section. The symbol table holds the information that the linker needs to bind multiple object files into a final executable file. An entry of the symbol table holds the following information: symbol name, binding (linkage scope: local or global), visibility (scope to other files: hidden, protected, or internal), and the virtual address of the symbol. Since every element has its own symbol name, we can get the virtual address of the symbol by looking up its name in the symbol table.

The last challenge is to find out how elements in an ELF file refer to each other at run time and how to find out such references. Searching all such references (where it is defined and where it is used) in a binary file is a daunting task without additional information. We found that we can obtain such references by using a linker option (*-q, or -emit-relocs*). This option produces relocation sections that include information about where the functions and variables are used in the program. Now we can gather all cross-reference information from the following sections: global offset table (.got), procedure linkage table (.plt), relocation data (.rel.data), and relocation text (.rel.text). The global offset table includes pointers to all of the static data in the program and the procedure linkage table stores pointers to all of the static code objects (functions) in the program. Therefore, these two sections provide the information about where the functions and variables are located in the program. Relocation sections such as .rel.text and .rel.data provide information about where the functions and variables are used in the program.

The rewriting process (user level permutation) comprises two major phases: 1) Coarse-grained permutation, and 2) Fine-grained permutation.

Coarse-grained Permutation The goal of the coarse-grained permutation is to shift code and data segments according to the given offset values from the user. Changing the order of code and data segments is also executed in this phase. To achieve the goal, the rewriting process goes through three stages: 1) ELF header rewriting, 2) Program header rewriting, and 3) Section rewriting.

Our tool first reads the ELF header to check if the target file is an ELF file. This can be done by reading the first four bytes of the file. If the file is an ELF object file, it should include the magic number in the *e_ident* member identifying itself as an ELF object format file. The tool then checks the sizes of the code and data segments to validate the given offset sizes from the user can fit into the user memory address space that are allowed in the Linux memory management scheme. Retrieving the location of the string table is then done. The string table holds information that represents all symbols, including section names referred in the program. Since each section header's *sh_name* member only holds an

[1]Further information about the sections and the ELF specifications can be obtained from a number of sources including [27, 28]

index to the string table, we need the string table during the entire rewriting process to look up the symbol name according to its index. The tool then rewrites the program entry point (*e_entry*), which is the virtual address where the system first transfers control to start the program, according to the offset value of the code segment.

Once we modified the ELF header, we need to change two entries in the program header: *p_vaddr* and *p_paddr*. They hold the virtual/physical addresses of the code and data segments that the loader needs to know for creating the program memory layout. Our tool modifies the *p_vaddr* and *p_paddr* values of the code and data segments according to the given offset values for the code and data segments.

Section rewriting is the most important stage in the coarse-grained permutation process to ensure a newly generated ELF file runs without any side effects (i.e., broken references). Since each section has different semantics and holds specific information, we need to know how to handle different sections and their entries. To take the case of symbol table section (.symtab), it holds all symbols used in both code and data segments. Therefore, if a symbol table entry refers a function in the code segment, we need to add the code segment offset value to the symbol's address value. Similarly, if an entry refers to a global variable in the data segment, we need to add the data segment's offset value.

Some sections require further understanding of how an ELF executable works during the run time to rewrite the program correctly. According to the ELF specification, some sections implicitly refer to other sections during the run time to resolve the symbol's actual address. For example, procedure linkage table (PLT) section contains a jump table used when the program calls functions during the run time. Since procedure linkage table entries store addresses pointing to the entries in the global offset table (GOT) to resolve actual addresses of the functions, we need to modify both PLT and GOT section entries together. Figure 1 shows the randomization example of the PLT and the GOT sections. A PLT entry at $0x804829c$ in figure 1(a) points the GOT entry that holds the jump address ($0x80482a2$). In figure 1(b), the procedure linkage table entry, related global offset table entry, and actual content of the GOT entry are modified after the randomization.

We also need to modify relocation sections since they hold information about where the functions and variables are referred to in the program. Rewriting entries in the relocation sections is done in a similar way figure 1.

Fine-grained Permutation The goal of fine-grained permutation is to randomly change the order of functions and variables within the code and data segments. By doing so, it brings additional protection against de-randomization attacks.

Fine-grained permutation comprises three stages: 1) Information Gathering, 2) Random Sequence Generation, and

```
0804826c <.plt>:
 804826c:       ff 35 b0 95 04 08          pushl  0x80495b0
   ...                ...                        ...
 8048297:       e9 d0 ff ff ff             jmp    804826c <.plt>
 804829c:       ff 25 c0 95 04 08          jmp    *0x80495c0
 80482a2:       68 10 00 00 00             push   $0x10

080495ac <_GLOBAL_OFFSET_TABLE_>:
 80495ac:       d0 94 04 08 00 00 00       rclb   0x8(%esp,%eax,1)
   ...                ...                        ...
 80495be:       04 08                      add    $0x8,%al
 80495c0:       a2 82 04 08 00             mov    %al,0x80482
 80495c5:       00 00                      add    %al,(%eax)
   ...
```

(a) Before permutation

```
0804c26c <.plt>:
 804c26c:       ff 35 b0 d5 05 08          pushl  0x805d5b0
   ...                ...                        ...
 804c297:       e9 d0 ff ff ff             jmp    804c26c <.plt>
 804c29c:       ff 25 c0 d5 05 08          jmp    *0x805d5c0
 804c2a2:       68 10 00 00 00             push   $0x10
 804c2a7:       e9 c0 ff ff ff             jmp    804c26c <.plt>

0805d5ac <_GLOBAL_OFFSET_TABLE_>:
 805d5ac:       d0 d4                      rcl    %ah
   ...                ...                        ...
 805d5bd:       c2 04 08                   ret    $0x804
 805d5c0:       a2 c2 04 08 00             mov    %al,0x804c2
 805d5c5:       00 00                      add    %al,(%eax)
```

(b) After permutation

Figure 1. PLT and GOT sections permutation

3) Entry Rewriting. The following information is gathered for the fine-grained permutation: section size, section's starting address, section's offset, total number of entries, the original order of entries, each entry's size, and each entry's starting address. The program header provides most of the information except for each entry's size and the entry's starting address. We can get each entry's starting address from the symbol table and calculate the entry's size by subtracting the address value of the current entry from the address of the next entry according to the original order of the entries. We store the gathered information in the data structure for later use.

We need to generate a randomized integer sequence to shuffle the functions and variables. To increase the randomness, we generate two separate randomized integer sequences for each code and data segment. We exploit the random() function and related functions provided by Linux operating system to generate the random sequences. The maximum number in the randomized sequence for code(data) segment is the same as the total number of entries of code (data) segment.

Entry rewriting is the last stage of fine-grained permutation. First, we rewrite the functions and variables according to the randomized sequences in a separate memory region. We then take out the original portions of the code and data segments from the ELF file and replace them with re-

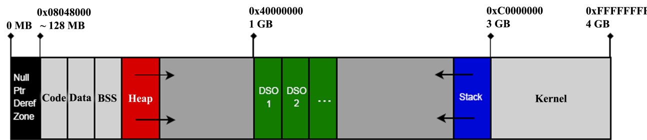

Figure 2. Normal process memory layout

arranged ones. Finally, we modify all the cross-references among functions and data objects. We change the relocation sections as shown in the coarse-grained permutation. We also modify offset values of all local function calls in the code segment. Local functions can only be called within the local scope (within the same file) and they are called by relative-offset from the current program counter (PC). For example, if the current PC is $0x8048000$ and the local function is located at $0x8049000$, then the offset used in the calling instruction is $0x1000$. We also change .rodata section that stores read-only (static) data objects, since control flow instructions (e.g., jmp or call) may refer to the values of the objects in the .rodata section to transfer the current control of the program.

Note that the protection scheme of fine-grained permutation is mainly dependent on the number of variables or functions in the program. If a program has few functions or global variables, the fine-grained permutation does not add strong protection on the code and data segments. However, if a program has a large number of functions and/or variables (e.g., Apache has over 900 variables), fine-grained permutation makes it difficult to guess correct locations of functions and variables.

3.2 Kernel Level Address Permutation

We build the ASLP kernel [32] for the popular 32-bit x86 CPU with Linux 2.4.31 kernel. Each process has its own virtual 32-bit address space ranging sequentially from 0 byte to 4 GB as shown in Figure 2. A program code segment starts from 0x8048000, which is approximately 128MB from the beginning of the address space. All data variables initialized by the user are placed in the data segment, and uninitialized data is stored in the bss segment. Shared libraries are placed in the dynamically shared objects (DSO) segments. The heap and the stack segments grow according to the user's request.

ASLP does not permute the top 1 GB of virtual address space (kernel space) since moving this region would require complex access control check on each memory access which introduces additional performance overhead. As a result, there are three regions for permutation: the user-mode stack, brk()-managed heap, and mmap allocations.

The User Stack The location of the user stack is determined and randomized during process creation. In the early stage of process creation, the kernel builds a data structure to hold process arguments and environment variables. This data structure is not yet allocated in the process memory, but rather it is prepared for the process in the kernel space memory. In this structure, the stack pointer is defined. This pointer is merely an offset into the first page of the soon-to-be stack region. We subtract a random amount between 0 and 4 KB from the stack pointer, thereby introducing randomization in low-order bits.

In the later stages of process creation, the same data structure is copied into the process address space. In this phase we introduce the large scale randomization. A random amount is subtracted from the standard 3 GB stack base location so that the region starts anywhere between approximately 128 MB and 3 GB.

To ensure the stack has room to grow, ASLP prevents subsequent allocations immediately below the stack. This feature prevents the stack from being allocated so close to another region that it cannot expand. The exact amount of reserved area is configurable, but our experiments show that 8 MB is sufficient for most applications.

The brk()-managed Heap Similar to the stack, the heap location is set during process creation. In an unmodified Linux kernel, the heap is allocated along with the BSS region, which is conceptually a part of the data segment. We modify the allocation code for the BSS and heap so they occur in two independent steps. Separation allows the heap location to be defined independently of the data segment. The amount of space to be allocated for the heap is then augmented by 4 KB (1 page). Then a random, page-aligned virtual address between 0 and 3 GB is generated for the start of the heap. Finally, a random value between 0 and 4 KB is added to this address to achieve sub-page randomization. Since the initial heap allocation was given an extra page, the sub-page shift will not push it beyond the original allocation. The heap also can grow to fulfill dynamic memory requirements as the corresponding process runs. As with the stack, a comparable solution is used for the heap in which an unused region of configurable size is maintained following the heap. This prevents the heap from being placed too close to other regions so that it has enough room to grow.

mmap() Allocations The mmap system call is used to map objects into memory. Such objects include shared libraries as well as any other files the application may wish to bring into memory. Allocations made by mmap are randomized using a one-phase, major randomization that is nearly identical to the primary phase used for the stack and heap. A secondary, sub-page shift is not used for mmap allocations, because doing so would violate the POSIX mmap specification [13]. Since there can be multiple independent mmap

allocations per process (such as for two different shared libraries), each allocation is made randomly throughout the entire available user level address space. This means that allocations for multiple shared libraries do not necessarily occupy a contiguous, sequential set of virtual memory addresses as they do in all related techniques and unrandomized kernels. This is beneficial because the location of one library will be of no use to determine the location of another library.

Although the mmap system call allows a user process to request a specific virtual address to store the mapping, there is no guarantee the request will be honored even in the vanilla kernel. In the ASLP kernel, if a specific address is requested it is simply disregarded and replaced by a random address. The random, page-aligned addresses are issued between 0 and 3GB. Therefore, mmap allocations use a one-phase major randomization rather than the two-phase approach used for the stack and heap. An exception to overriding supplied addresses exists for fixed regions, such as the code and data segments. These regions are also brought into memory via mmap, but because they are flagged as fixed the supplied address is honored without modification.

3.3 Demonstration of Permutation

After both user level and kernel level permutations, all critical memory regions including static code and data segments can be placed in different locations throughout the user memory space. Figure 3 shows a possible permutation of the normal process memory layout as shown in figure 2. The heap is allocated independently of the data segment, and the stack is not the highest allocation in the user space. The data segment comes first instead of the code segment. In short, the static code, data, stack, heap, and mmap allocations occur randomly throughout the 3 GB user address space. Figure 4 shows an example of fine-grained permutation. Global variables (from *num1* to *num6*) are randomly re-ordered after the permutation.

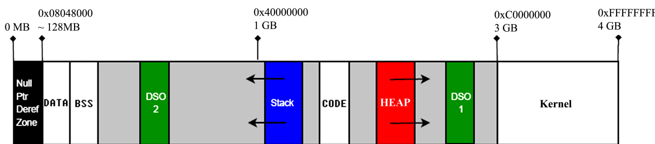

Figure 3. Coarse-grained permutation

```
Disassembly of section .data:
8049684 <__data_start>:
8049684:    00 00            add   %al,(%eax)
8049688 <__dso_handle>:
8049688:    00 00            add   %al,(%eax)
804968c <p.0>:
804968c:    9c               pushf
804968d:    95               xchg  %eax,%ebp
804968e:    04 08            add   $0x8,%al
8049690 <num1>:
8049690:    01 00            add   %eax,(%eax)
8049694 <num2>:
8049694:    02 00            add   (%eax),%al
8049698 <num3>:
8049698:    03 00            add   (%eax),%eax
804969c <num4>:
804969c:    04 00            add   $0x0,%al
80496a0 <num5>:
80496a0:    05 00 00 00 06   add   $0x6000000,%eax
80496a4 <num6>:
80496a4:    06               push  %es
80496a5:    00 00            add   %al,(%eax)
```

```
Disassembly of section .data:
8049684 <__data_start>:
8049684:    00 00            add   %al,(%eax)
8049688 <__dso_handle>:
8049688:    00 00            add   %al,(%eax)
804968c <p.0>:
804968c:    9c               pushf
804968d:    95               xchg  %eax,%ebp
804968e:    04 08            add   $0x8,%al
8049690 <num4>:
8049690:    04 00            add   $0x0,%al
8049694 <num5>:
8049694:    05 00 00 00 06   add   $0x6000000,%eax
8049698 <num6>:
8049698:    06               push  %es
8049699:    00 00            add   %al,(%eax)
804969c <num2>:
804969c:    02 00            add   (%eax),%al
80496a0 <num1>:
80496a0:    01 00            add   %eax,(%eax)
80496a4 <num3>:
80496a4:    03 00            add   (%eax),%eax
```

Figure 4. Fine-grained permutation

Region	Vanilla	Exec-Shield	PaX ASLR	ASLP
User Stack	0 bits	17	24	28
Heap	0 bits	13	13	29
Mmap	0 bits	12	16	20
Code	0 bits	0	0	20
Data	0 bits	0	0	20

Table 2. PaXtest results

4 Evaluation

4.1 Security Evaluation

As discussed previously, although every address in the x86 memory architecture is represented by 32-bits, not all of those bits can be randomized. To assess the bits of randomness in each region and for each technique, we use a third-party application called PaXtest [5]. PaXtest includes two types of tests. First, there are tests that try overwriting tiny executable code (e.g., just ret) in different memory regions (stack, heap, bss, data, anonymous) and check if the executable code can be run on the region. If the code works, it reports that the system is vulnerable on such region. The second type of tests measures the randomness of each region. By running locally, PaXtest is able to spawn simple helper programs that merely output the virtual addresses of their variables in a randomized region. In doing so, it can determine the number of randomized bits for such variables based on where they would normally reside in an unrandomized system. Table 2 provides a summary of the PaXtest results.

The vanilla kernel has no randomization. So each region has zero bit of randomness. Exec-Shield is the most susceptible to de-randomization attacks, providing the lowest amount of randomization in all five regions. PaX ASLR comes in the middle, with significantly better stack and mmap randomization. ASLP comes out ahead by at least 4 bits in all regions and provides randomization on the static code and data segments with 20 bit randomness.

We used the PaXtest results to estimate the probabilistic protection provided by ASLP. For example, ASLP randomize 20 bits in the address of the shared libraries. This means that there are 2^{20} possible locations for placement for the

shared libraries. Assuming a random distribution, the address can be guessed in a number of guesses equal to half of the possible locations. Knowing both the reported attack duration of 216 seconds to brute-force guess the address of the shared library region [21] and the number of possible locations in this region from Table 2, we can estimate the average guess rate. Equation 1 shows the calculation of guess rate.

$$\frac{2^{20}\,_{\text{possible locations}}}{2} \rightarrow \frac{524288\,_{\text{average guesses}}}{216\,_{\text{seconds}}} = 2427.2\,_{\text{seconds to guess on average}} \quad (1)$$

Using the rate of 2427.2 guesses per second derived from equation 1 and the bits of randomness returned from PaX-test in Table 2, we can calculate the amount of time required to brute force the randomization in each memory region for ASLP. It takes about 3 weeks to guess correct location of heap by brute force searching. The stack takes 10 days to de-randomize. Mmap, code, and data regions cause lowest amount of time–about an hour. However, code and data regions would take additional time to brute force due to the fine-grained randomization of functions and variables.

Against ASLP, a de-randomizing attack would take a considerable amount of time for constant guessing. However, we do not claim that ASLP is an effective deterrent to prevent a determined attacker from penetrating a single randomized target. In such micro-security level perspective, the de-randomizing attack by Shacham et al [21] can still succeed in about an hour to guess the mmap region. Instead, we argue from a macro-security perspective that ASLP provides a mechanism by which the memory corruption attack on large machines can be slowed to a rate that allows intrusion detection systems and system administrators to respond.

Consider the effect of address randomization at the macro-security level: a large scale Internet worm propagation. With ASLP, the speed at which worms can spread using memory corruption attacks is bounded not by how fast they can reach vulnerable hosts, but by how fast they can de-randomize the randomized address space of each target.

Ideally, the current fastest known spreading worm, The Sapphire/Slammer[17], is able to infect 100% of vulnerable hosts in less than a minute by doubling in size every one second with no randomization address space. We calculate the worm propagation rate of each randomization technique based on the infection rate of Sapphire/Slammer worm and the probabilistic protection of each technique discussed above. For Exec-Shield, 100% infection occurs in just over four minutes (4.275 minutes); for PaX ASLR the time is just over one hour (68.4 minutes). Our ASLP approach is able to delay 100% infection for over eighteen hours (1,083 minutes). This extension of infection time illustrates the benefit of having effective address space layout protection because fast worms that exploit memory corruption vulnerabilities must first get through address randomization before they can compromise their targets. Further, the barrage of attempts to guess the correct address should be visible by intrusion detection and prevention systems. Increasing probabilistic protection means forcing attackers to make more guesses, effectively giving intrusion detection systems a bigger target.

4.2 Performance Evaluation

Our goal is to incur comparable or less performance overhead to related techniques. We compare our approach with two other popular ones: PaX ASLR and Exec-Shield. It should be noted that both PaX ASLR and Exec-Shield can be configured to do more than address randomization. Where possible, we disable their extra features to make them as similar to ASLP as possible.

The various kernels are tested by installing each of them on a single test computer and selecting the desired kernel at boot time. The test computer runs Red Hat Linux 9.0 with a 2.66 GHz Intel Pentium 4 CPU, 512 MB of RAM, and a 7200 RPM ATA IDE hard disk drive. All benchmark suites were compiled using GCC version 3.4.4.

The vanilla kernel establishes the baseline by which the absolute performance impact of other configurations can be measured. Since PaX ASLR and Exec-Shield are closely related works that provide kernel-based address layout randomization, their performance provides a metric to determine if ASLP does or does not have comparable performance with kernel level randomization. For user level randomization, we compare ASLP with Position Independent Executable(PIE). PIE is used in PaX ASLR to move the static code and data regions from their traditionally fixed positions. Since PIE is a compilation option and not a kernel modification, we test it using a vanilla kernel and compile the benchmark suite with the PIE option.

We employ three popular benchmarks to measure the performance of each configuration: SPEC CPU2000[7], LMBench micro-benchmark[15], and Apache Benchmark[11].

SPEC CPU2000 Benchmark The SPEC CPU2000 Integer benchmark suite is a set of computationally intensive integer applications that simulate the workload of scientific computation. Each benchmark in the CPU2000 suite measures the time required to complete a different integer computation. For this reason, we use the CPU2000 benchmark to measure the impact of address permutation on computational efficiency. Table 3 gives the result of CPU2000 Integer benchmark[2].

[2] The full CPU2000 Integer benchmark suite contains 12 benchmarks. One of the 12, the "eon" benchmark, did not compile on GCC version 3.4.4. Although this GCC version was newer than that recommended by the benchmark documentation, a GCC compiler of at least version 3.4.4

Benchmark	Vanilla	Exec-Shield	PaX ASLR	PIE	ASLP
gzip	177	178	176	207	177
vpr	284	284	284	298	284
gcc	132	134	131	148	133
mcf	408	413	409	427	410
crafty	116	116	117	142	116
parser	266	268	266	281	267
perlbmk	168	166	166	248	166
gap	126	128	125	144	128
vortex	185	185	185	218	187
bzip2	260	257	257	285	256
twolf	514	505	506	525	504
Total	2636	2634	2619	2923	2628
Avg. Overhead(%)	0	0.14	-0.55	14.38	-0.3

Table 3. SPEC CPU2000 benchmark run times (seconds)

For the kernel level randomization comparison, Exec-Shield has an average performance overhead of 0.14%, but PaX ASLR does not incur additional overhead. For the user level randomization comparison, PIE shows 14.38% average performance overhead. The overhead of PIE mainly comes from additional instructions that need to resolve actual memory addresses of program objects during the run time. Recent work by Bhatkar et al. in [3] also took a similar indirection approach to randomize static code and data segments. Such indirection causes an average runtime overhead of about 11% in their experiments. ASLP shows no performance overhead which supports our claim that ASLP has computational performance overhead comparable to closely related works.

LMBench benchmark The LMBench benchmark suite differs from CPU2000 because it strives to benchmark general operating system performance rather than the computational performance of a set of applications. The LMBench operating system benchmark suite consists of five sets of micro-benchmarks, each of which is designed to focus on a specific aspect of operating system performance. We only consider kernel level permutation in this evaluation since LMBench benchmark targets at an operating system's performance overhead rather than an application's performance overhead. The result shows that the process creation overhead is the primary source of expected overhead from address space randomization techniques like PaX ASLR, Exec-Shield, and ASLP, because additional operations are inserted into the process creation functions. ASLP slows down fork() and exec() operations by 6.86% and 12.53% respectively. Both PaX ASLR and Exec-Shield have consistently higher overheads for the same tasks: 13.83-21.96% and 12.63-32.18% respectively. The context switching overhead results give a similar story to process operation results. ASLP caused 3.57% which is less than Exec-Shield and PaX ASLR. Their results are 8.15% and 13.8% respectively. In File and virtual memory (VM) system latency results, ASLP and PaX ASLR incur 12% overhead for mmap latency due to the increasing number of instructions to complete mmap allocations. However, average overheads of file and VM operations are very low in all three techniques. We do not consider local communication overhead from LMBench benchmark result, since we made no changes to networking code. Instead, we have the Apache benchmark to evaluate remote communication performance overhead.

was needed to support PIE linking. We therefore elected to exclude the eon benchmark from the final results.

Apache Benchmark The Apache Benchmark [11] measures the performance of an Apache HTTP server [12] in terms of how quickly it can serve a given number of HTML pages via TCP/IP. Our Apache Benchmark configuration makes 1 million requests, in simultaneous batches of 100, for a static HTML page of 1881 bytes, which includes 425 bytes of images. The result shows that only PIE incurs major overhead, about 14%. The other techniques, including ASLP, shows less than 1% overhead.

5 Limitations

Although ASLP can mitigate many types of attacks, current implementation of ASLP does not support stack frame randomization. Lack of randomization on the stack frame allows attackers to exploit a return-to-libc attack as described in [21]. We can mitigate such attack by adding pads among elements in the stack [2], but the randomization is limited and it wastes memory spaces. Further investigation is required to find a better solution. Another limitation is that ASLP might require re-linking or recompilation of source codes if a program (executable binary) does not have relocation information. Although our goal is to perform randomization without source code access, current implementation requires the relocation data from the compiler. However, we believe that this is a one time effort for life time benefit, since once the relocation data is included, we can randomize the program repeatedly to thwart real-life attacks.

6 Conclusions

This paper investigated methods for improving the address space randomization techniques for the purpose of increasing resistance to memory corruption attacks. ASLP provides both user and kernel level randomizations. We developed a novel binary rewriting tool that allows users to permute static code and data regions with fine-grained level. We also modified the Linux operating system kernel to provide system wide randomization protection. Combined together, ASLP permutes the program's memory layout completely within 3 GB user space memory. Using various benchmarks we found that it is possible to achieve better randomization for process virtual memory layout with-

out incurring obtrusive performance overhead. The performance overhead of ASLP is low as compared with other address space randomization techniques that provide less randomization. With ASLP, runtime overhead is less than 1%, as indicated by both the SPEC CPU2000 Integer Benchmark as well as the Apache Benchmark.

This paper also validates the use of address space randomization by demonstrating that the randomization provided by ASLP dramatically reduces the speed at which worms can propagate throughout the Internet. The increase in time needed to exploit targets introduced by randomization means that the fastest infection time for worms relying on the absolute location of either the user stack, heap, or an mmap allocation is on the order of hours, not minutes.

References

[1] Anonymous. Once upon a free(). Phrack Magazine, 11(57), August 2001. Available from URL http://www.phrack.org/phrack/57/p57-0x09.

[2] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an effcient approach to combat a broad range of memory error exploits. In V. Paxson, editor, Proceedings of the 12th USENIX Security Symposium, pages 1020, August 2003.

[3] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, pages 271-286, July 2005.

[4] Joshua Brindle. Hardened gentoo. Available from URL http://www.gentoo.org/proj/en/hardened/.

[5] Peter Busser. Paxtest. Available from URL http://www.adamantix.org/paxtest/.

[6] The Fedora Community. The fedora project. Available from URL http://fedora.redhat.com/.

[7] Standard Performance Evaluation Corporation. Spec cpu2000 v1.2. Available from URL http://www.spec.org/cpu2000/.

[8] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In 7th USENIX Security Symposium, San Antonio, Texas, January 1998. Available from URL http://wirex.com/ crispin/usenixsc98.pdf.

[9] Ulrich Drepper. Security enhancements in red hat enterprise linux (besides selinux). Available from URL http://people.redhat.com/drepper/nonselsec.pdf, June 2004.

[10] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In 6th Workshop on Hot Topics in Operating Systems, Los Alamitos, CA, pages 62-72. IEEE Computer Society Press, 1997.

[11] Apache Software Foundation. Apache benchmark. Available from URL http://httpd.apache.org/docs/2.0/programs/ab.html.

[12] Apache Software Foundation. Apache http server project. Available from URL http://httpd.apache.org.

[13] The IEEE and The Open Group. The Open Group Base Specifications: mmap, 6 edition, 2004. Availale from URL http://www.opengroup.org/onlinepubs/009695399/functions/mmap.html.

[14] The IEEE and The Open Group. The Open Group Base Specifications: rand, 6 edition, 2004. Availale from URL http://www.opengroup.org/onlinepubs/009695399/functions/rand.htm.

[15] Larry McVoy and Carl Staelin. Lmbench: Tools for performance analysis. Available from URL http://www.bitmover.com/lmbench/.

[16] Ingo Molnar. Exec-shield. Available from URL http://people.redhat.com/mingo/exec-shield/.

[17] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. The spread of the sapphire/slammer worm, 2003. Available from URL http://www.cs.berkeley.edu/ nweaver/sapphire/.

[18] George Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Portland, Oregon, pages 128-139, January 2002.

[19] Aleph One. Smashing the stack for fun and profit. Phrack Magazine, 49(14), November 1996. Available from URL http://www.phrack.org/phrack/49/P49-14.

[20] Scut. Exploiting format string vulnerabilities, March 2001. Available from URL http://julianor.tripod.com/teso-fs1-1.pdf.

[21] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In V. Atluri, B. Pfitzmann, and P. McDaniel, editors, Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, D.C. ACM, October 2004.

[22] The PaX Team. The pax project, 2001. Available from URL http://pax.grsecurity.net/.

[23] The PaX Team. Address space layout randomization, March 2003. Available from URL http://pax.grsecurity.net/docs/aslr.txt.

[24] United States Computer Emergency Readiness Team (US-CERT). Technical cyber security alerts. Available from URL http://www.us-cert.gov/cas/techalerts/.

[25] Tony Warnock. Random-number generators. Los Alamos Science, 1987. Available from URL http://www.fas.org/sgp/othergov/doe/lanl/pubs/00418729.pdf.

[26] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In A. Fantechi, editor, Proceedings of the 22nd Sympomsium on Reliable Distributed Systems, pages 260-269. IEEE Computer Society, October 2003.

[27] Mary Lou Nohr. Understanding ELF Object Files and Debugging Tools. Number ISBN: 0-13-091109-7. Prentice Hall Computer Books, 1993.

[28] Tool Interface Standard (TIS) Committee. Executable and Linking Format (ELF) Specification, 1995.

[29] S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for C program analysis and transformation. In Conference on Compiler Construction, 2002.

[30] SecuriTeam. Overwriting ELF .dtors section to modify program execution. Available from http://www.securiteam.com/unixfocus/6H00I150LE.html.

[31] SecurityTracker. Advisory 16/2005: phpMyAdmin Local File Inclusion. Available from http://securitytracker.com/alerts/2005/Oct/1015091.html

[32] Christopher G. Bookholt. Address Space Layout Permutation:Increasing Resistance to Memory Corruption Attacks. M.S. thesis, North Carolina State University, 2005.