# A Practical Approach for Adaptive Data Structure Layout Randomization

Ping Chen[1,2,3], Jun Xu[1], Zhiqiang Lin[4], Dongyan Xu[3], Bing Mao[2], and Peng Liu[1]

[1]College of Information Sciences and Technology, The Pennsylvania State University
[2]State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
[3]Department of Computer Science, Purdue University
[4]University of Texas at Dallas
{pzc10,jxx13,pliu}@ist.psu.edu, zhiqiang.lin@utdallas.edu, dxu@cs.purdue.edu, maobing@nju.edu.cn

**Abstract.** Attackers often corrupt data structures to compromise software systems. As a countermeasure, data structure layout randomization has been proposed. Unfortunately, existing techniques require manual designation of randomize-able data structures without guaranteeing the correctness and keep the layout unchanged at runtime. We present a system, called SALADS, that automatically translates a program to a DSSR (Data Structure Self-Randomizing) program. At runtime, a DSSR program dynamically randomizes the layout of each security-sensitive data structure by itself autonomously. DSSR programs regularly re-randomize a data structure when it has been accessed several times after last randomization. More importantly, DSSR programs automatically determine the randomizability of instances and randomize each instance independently. We have implemented SALADS based on gcc-4.5.0 and generated DSSR user-level applications, OS kernels, and hypervisors. Our experiments show that the DSSR programs can defeat a wide range of attacks, and the performance overhead is acceptable.

## 1 Introduction

In programs developed in `C` or `C++` language, encapsulated data objects, such as `struct` and `class`, are widely defined to group a list of logically related variables. In this paper, we denote these encapsulated data objects as data structures[1]. Not surprisingly, data structures are in the meantime the target or aid of a wide variety of attacks. Attackers often leverage knowledge about data structures defined in a victim program to construct successful exploits against it. This is the case for both application programs and system programs (e.g., operating system kernels and virtual machine monitors). A data structure contains a set of fields.

[1] In a broad sense, a data structure means a particular way of organizing data in a program so that it can be used efficiently. In this paper, we narrow down the scope of data structures to be encapsulated data objects in `C` or `C++` programs.

Knowledge about a data structure's layout, namely how the fields neighbour each other inside the data structure, can be very useful to the attacker. For example, knowing the layout of accounting/book-keeping data structures, on-line gaming fraud [11] can be performed by modifying the values of relevant fields; Knowing the layouts of in-stack or in-heap data structures will help construct memory corruption exploits (e.g., privilege escalation attacks against openssh-2.1.1 [14]); Guided by the layout of the process control block (PCB), a kernel rootkit is able to hide a process by locating and manipulating certain pointer fields. We define attacks that locate a data structure and manipulate specific fields after knowing its layout as *data structure manipulation* attacks.

Randomizing either the location or the layout of the target data structure will significantly raise the bar for data structure manipulation attacks. There have been two lines of research towards achieving such randomizing goals: (1) Address Space Layout Randomization (ASLR) randomly arranges the base addresses of segments (e.g., stack), which has been widely researched and deployed. Recently, fine-grained ASLR techniques have been proposed to achieve randomization at different levels, including page level [6], function level [23], basic block level [28], and instruction level [38,20]. (2) Data Structure Layout Randomization (DSLR) [25,34] reorders the fields or inserts dummy fields in encapsulated data objects (e.g., `struct`). With DSLR deployed, the layouts of data structures are randomized to break the mono-culture of programs.

However, ASLR or fine-grained ASLR techniques have two limitations: (1) ASLR is vulnerable to memory content leakage [33,12,35,31,22,10,30,43]. By leveraging memory contents leakage, an attacker can infer the base addresses of memory regions (e.g., segments or pages) under ASLR. Knowing the offset of the target data structure in the containing region[2], the attacker can figure out its base address. (2) ASLR can be easily circumvented by rootkits, such as those leveraging Direct Kernel Object Manipulation (DKOM) [29,8,22]. In many cases, a rootkit knows the base address of the target data structure even if ASLR is deployed. For example, kernel global data structures can be located by referring to kernel symbols (i.e., `/proc/kallsyms`). In other cases, a rootkit has no such knowledge, but it has the privilege to read arbitrary memory and thus can infer such a base address[2].

In this paper, we present a novel technique, adaptive DSLR, to defend against data structure manipulation attacks. We design a compiler-based system, called SALADS[3], to implement our technique. SALADS transforms a program into a Data Structure Self-Randomizing (DSSR) program. A DSSR program periodically re-randomizes a data structure after the data structure has been accessed for a certain number of times since last re-randomization. The re-randomization is independently and asynchronously performed on each instance even if they have the same data structure definition. To avoid errors (e.g., pointer reference corruption), SALADS automatically determines the randomizability of data structure

[2] By simulating the execution in local environment without ASLR, the attacker can infer the offsets from the target data structures to a particular memory region.

[3] SALADS stands for Self Adaptation of LAyout of Data Structures.

instances without programmer's input and de-randomizes a data structure that might have been unsafely randomized.

SALADS can address the two limitations of ASLR: suppose the base address of the target data structure is exposed when memory content leakage happens or when a rootkit is launched. The layout of the data structure is randomized when SALADS is deployed. Therefore, the attacker in general cannot accurately locate specific fields. Even if the attacker could infer the current layout of the target data structure, the attacker could be stopped by the adaptation (i.e., dynamic self-re-randomization). In one attack, the layout inferring part and the data structure manipulation part are typically completed in chronological order. After the layout inferring but before the data structure manipulation, DSSR programs may have already re-randomized the target data structure. Consequently, the attacker would mistakenly manipulate irrelevant fields.

We refer the existing DSLR technique [25,34] as static DSLR. Compared with static DSLR, our adaptive DSLR offers several unique features: (1) Instead of randomizing data structures layout at compile-time/load-time, DSSR programs generated by SALADS re-randomize data structures at runtime. Without this feature, static DSLR shares the two limitations with ASLR. When memory content leakage happens or when a rootkit is launched, the randomized layout of the target data structure can be reverse engineered (e.g., [7]). Examples of how to reverse engineer the layout are presented in Section 2. Once the layout of the target data structure is inferred, the attacker could correctly manipulate specific fields. (2) A DSSR program randomizes each data structure instance independently and asynchronously, regardless of their types. Without this feature, static DSLR can be circumvented in situations where the target data structure is not initialized. For example, rootkits can speculate the layout of the target data structure instance by referring another initialized instance of the same type. In a kernel with static DSLR deployed, the layout inferred in such a way enables the rootkits to successfully manipulate the expected fields. (3) In case an instance is involved in a statement that might cause inconsistency or crash, the DSSR program will restore the instance to its original layout. The restoring process is denoted as de-randomization.

Our contributions in this work are as follows:

- This is the first effort toward runtime adaptive DSLR.
- Our experiments show that DSSR programs are able to address the two limitations of ASLR in thwarting data structure manipulation attacks.
- DSSR programs generated by SALADS automatically determine the randomizability of data structures without programmers' assistance. Instances determined to be un-randomizable will be de-randomized to avoid errors.
- SALADS achieves both cross instance diversity (different randomized layouts for different instances of the same type) and cross time diversity.
- On average, the performance overhead introduced by SALADS is (1) 6.3% for application programs (randomly selecting 20% of data structures to protect in SPECInt2000, `httpd-2.0.6`, `openssh-2.1.1p4`, and `openssl-0.9.6d`); (2) 16.7% for OS kernels (selecting 23 security-sensitive data structures

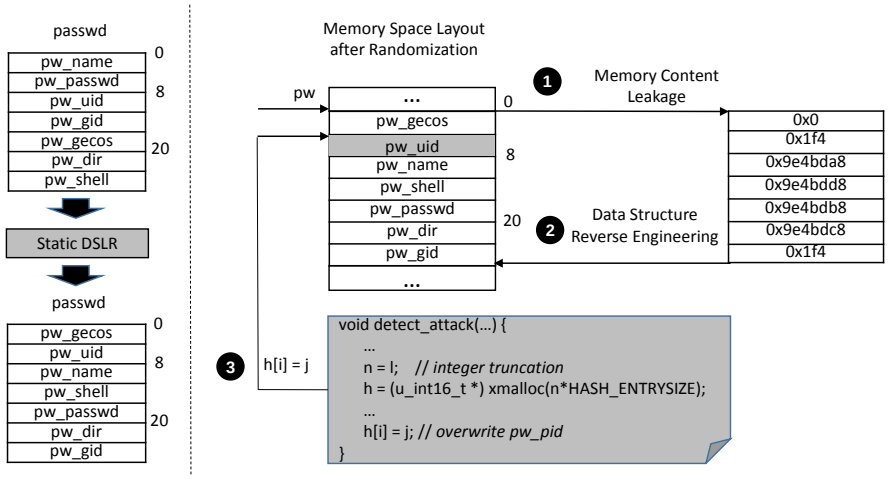



**Fig. 1.** Privilege escalation in openssh under ASLR and static DSLR

to protect in Linux kernel); (3) 4.5% for hypervisor (selecting 20 security-sensitive data structures to protect in Xen hypervisor).

## 2 Overview

### 2.1 Threat Model

Our threat model focuses on data structure manipulation attacks. We subdivide such an attack into three steps: (Step-I) attacker gets the memory location of a data structure instance; (Step-II) attacker figures out its layout; (Step-III) attacker reads/writes certain fields of the instance.

**Data Structure Manipulation with Memory Content Leakage in Applications.** We take the privilege escalation attack against openssh-2.1.1 (CVE-2001-0144) [14] as an illustrating example. The goal of the attack is to modify the field `pw_uid` in the instance `pw` (of type `struct passwd`) to escalate the remote shell with root privilege. The three steps in this attack are as follows. First, the attacker gets to know the base address of `pw`; Second, the attacker figures out the layout of `pw`; Third, the attacker writes the maliciously-crafted value to `pw->pw_uid` by exploiting an integer truncation bug.

In Figure 1, we present how to conduct the above privilege escalation attack under ASLR and static DSLR (the base address and the layout of `pw` are both randomized). At Step-I, an attacker can resort to memory content leakage (e.g., memory disclosure [33], uninitialized memory tracking [12], side channel [10,30,43]) (❶). Assuming the attacker has obtained the disclosed memory page that contains `pw`, he/she can search the signature of `struct passwd`[4] in the page.

[4] Inside `struct passwd`, `pw_uid` and `pw_gid` are identify numbers with small values ( $\leq 0xFFFF$); `pw_passwd`, `pw_name`, `pw_shell`, and `pw_dir` are four pointers and their values form an arithmetic progression with common difference of 16; `pw_gecos` is 0.

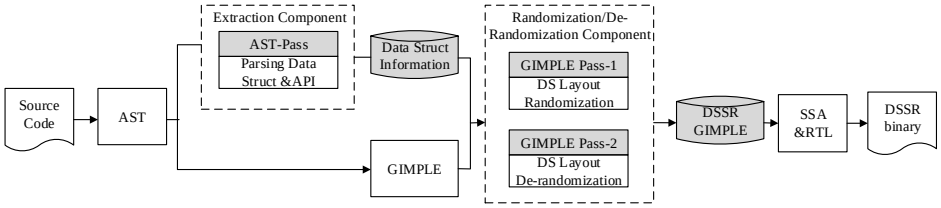



**Fig. 2.** System Overview

If the search succeeds, the attacker can locate the base address of `pw`. At Step-II, the attacker can reverse engineer the contents of `pw` to recover locations of specific fields (e.g., `pw_uid` and `pw_gid` have unique values[4]) (❷). Since ASLR and static DSLR do not randomize `pw` at runtime, the attacker can correctly modify `pw_uid` and `pw_gid` (❸) to escalate the privilege.

**Data Structure Manipulation by Rootkits under ASLR and static D-SLR.** Many rootkits achieve their goals via manipulating data structures, such as the one presented in [21]. However, ASLR and static DSLR make such manipulation more difficult (by randomizing the base address and the layout of the target data structure). In Section 1, we have explained how a rootkit can bypass ASLR and static DSLR. For instance, `taskigt` is a rootkit that stealthily promotes privileges of a process when the process opens a specific `proc` file. The rootkit attempts to modify a local data structure instance `proc_ent` of type `proc_dir_entry`. Most fields in `proc_ent` are not initialized, including the target field `read_proc` (a function pointer). The rootkit can infer `read_proc` in a global variable `proc_root` of type `proc_dir_entry` by reverse engineering (most fields in `proc_root` are initialized). In this way, the rootkit can locate `read_proc` in `proc_ent` (`read_proc` in `proc_ent` and `proc_root` have the same offset). Then the rootkit manipulates `read_proc`, to make it point to a malicious function.

### 2.2 System Overview

**Key Idea.** By breaking any of the three steps in the threat model, we would be able to defeat a data structure manipulation attack. However, since it is hard to eliminate memory content leakage and rootkits, attackers can succeed at Step-I and Step-II even if modern defenses are deployed. Our idea is to disrupt Step-III. Specifically, we adaptively randomize layout of each data structure instance independently at runtime. The key is that the target instance might be re-randomized between Step-II and Step-III. Therefore, the attacker may not accurately access the targeted fields.

**Compilation Steps.** We design SALADS to realize the above idea. SALADS is built on top of the GNU GCC compiler. Figure 2 shows the compilation steps of SALADS, with the white boxes indicating the original GCC compilation phases. As shown in the figure, SALADS adds two key components to GCC: the extraction component and the randomization/de-randomization component. We briefly explain the compilation steps as follows. (1) SALADS parses

the source code into an Abstract Syntax Tree (AST). (2) The extraction component (i.e., AST-Pass) traverses the AST to collect required information for the randomization/de-randomization component. (3) SALADS transforms the AST into the GIMPLE representation. (4) The randomization component (i.e., GIMPLE Pass-1) replaces each statement that accesses data structures with DSSR statements. These DSSR statements randomize/re-randomize the layout of the accessed data structures at runtime. (5) The de-randomization component (i.e., GIMPLE Pass-2) inserts de-randomizing statements before each dangerous statement to de-randomize involved data structures. We will explain which statements are dangerous later. (6) SALADS compiles the GIMPLE representation into a DSSR binary in remaining phases (e.g., SSA, RTL). The DSSR binary can self-rerandomize/de-randomize data structure instances at runtime.

## 3 Design and Implementation of SALADS

### 3.1 Extraction Component

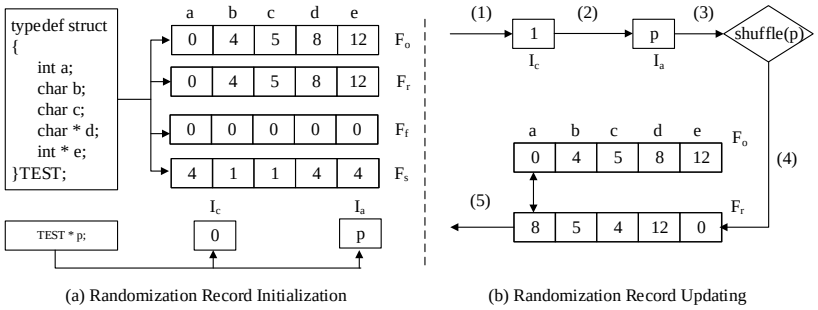
As shown in Figure 2, the extraction component is designed to gather definitions of data structures and usages of external & shared APIs. The gathered information is later used by the randomization/de-randomization component.

**Extracting Data Structures.** For each definition of data structure encountered during AST traversal, the extraction component records the name of the data structure as well as the name, the size and the offset of each field. To calculate the offset or size of a field in a data structure, two challenges need to be tackled. The first one is the alignment. A compiler often allocates fields in a data structure on aligned boundaries [39]. In our design, the extracting component calculates the offsets of fields based on the compiling options specified by the programmers (e.g., #param pack(n)). If no such options are available, the extracting component relies on the default alignment rules to redress the offsets. The second challenge is how to handle arrays with flexible sizes [1]. If a field is an array with flexible size, its size cannot be determined by the compiler. In such a case, SALADS can only arrange this field to the end of the data structure. Correspondingly, SALADS will mark this field as un-randomizable.

**Identifying External and Shared APIs.** External APIs refer to functions that are used but not defined in a program. The extracting component records usage of all external APIs. For a program, shared APIs are functions defined in this program but publicly used by other programs. For instance, system calls are shared APIs in the Linux kernel. Identifying shared APIs in a program requires knowledge about which functions defined in this program are publicly used by other programs. Such knowledge is often well documented.

### 3.2 Randomization Component

The randomization component (i.e., GIMPLE Pass-1) instruments the GIMPLE representation. The instrumented program can self-randomize the layout of data




(a) Randomization Record Initialization (b) Randomization Record Updating

**Fig. 3.** Initialization and updating of a randomization record

structures at runtime. The instrumentation replaces each statement that contains data structure accesses with a set of DSSR statements, details of which are presented next.

**Data Structure Layout Randomization.** First, GIMPLE Pass-1 iterates statements in the GIMPLE representation. Second, the pass parses each statement to identify data structure field accesses. For each field access, the pass inserts the DSSR statements before the containing statement. The DSSR statements firstly randomize the layout of the instance. Afterwards, if the access is a read, the DSSR statements maintain the value of the accessed field in a temporary variable. If the access is a write, the DSSR statements use a temporary pointer to point to the after-randomized location of the accessed field. Finally the pass replaces the parsed statement with a new statement. In the new statement, each data structure field access is replaced with the corresponding temporary pointer or the temporary variable.

A statement is parsed as follows. First, the statement is parsed into expressions in a right-to-left order. If an expression is compound (e.g., `a+b`), it will be decomposed into atomic expressions (e.g., `a` and `b`). If an atomic expression is a data structure field access, the parser records the type of the instance, the address of the instance, and the name of the field. In particular, the data structure field access could be nested. For instance, `A->B.x` involves two nested accesses: `A->B` and `B.x`. In such case, the parser firstly parses the outer access and then parses the inner access. In the example of `A->B.x`, `A->B` is processed at first and `B.x` is processed next.

The DSSR statements insertion for data structure field accesses follows the same order as they are parsed. For an access, the inserted DSSR statements include: (1) a gimple statement to invoke the `Initialize_Record` routine. The routine first checks whether this instance is recorded. If not, it initializes a `randomization record`. The randomization record contains following metadata of the instance: $I_a$ (memory address of this instance), $I_c$ (how many times the instance has been accessed since last randomization). A randomization record also maintains the metadata for each field in the instance: $F_o$ (original offset), $F_r$ (after-randomized offset), $F_s$ (size), and $F_f$ (randomization flag). The randomization flag indicates whether a field is randomizable; (2) a GIMPLE statement to invoke the `Update_Record` routine. This routine increases $I_c$ by 1 and then

```
1  typedef struct{
2      int a;
3      char b;
4      char c;
5      char * d;
6      int * e;
7  }  TEST;
8
9  void main()
10 {
11     TEST * p;
12     p = (TEST *)malloc
           (sizeof(TEST));
13     TEST q;
14     p->a = 1;
15     q.c = 'a';
16 }
```

(a) Source Code

```
1  main()
2  {
3      void * D.1962;
4      struct TEST * p;
5      extern void * malloc
                (unsigned int);
6      struct TEST q;
7      D.1962 = malloc(16);
8      p = (struct TEST *)
                        D.1962;
9      p->a = 1;
10     q.c = 97B;
11 }
```

(b) GIMPLE Output by GCC-4.5.0

```
1  main()                          16     p = (struct TEST *) D.2052;
2  {                               17     p.0 = (int)p;
3      void * D.2052;              18     Initialize_Record(0,p.0);
4      int p.0;                    19     Update_Record(0,p.0);
5      int * D.2054;               20     D.2057 = Offset_Diff(0,1);
6      int q.1;                    21     D.2054 = &p->a;
7      char * D.2056;              22     D.2058 = D.2054 + D.2057;
8      struct TEST * p;            23     *D.2058 = 1;
9      extern void * malloc        24     q.1 = (int) &q;
            (unsigned int);        25     Initialize_Record(0,q.1);
10     struct TEST q;              26     Update_Record(0,q.1);
11     int D.2057;                 27     D.2059 = Offset_Diff(0,3);
12     int * D.2058;               28     D.2056 = (char *)&q.c;
13     int D.2059;                 29     D.2060 = D.2056 + D.2059;
14     char * D.2060;              30     *D.2060 = 97B;
15     D.2052 = malloc(16);        31 }
```

(c) DSSR GIMPLE Output by SALADS

**Fig. 4.** An example showing how DSSR program generated by SALADS works

checks whether $I_c$ exceeds a threshold $W_m$. If so, this routine randomly shuffles the fields in the memory space of the data structure and records the after-randomized offsets into $F_r$; (3) a GIMPLE statement to call the `Offset_Diff` routine for calculating the offset difference between the randomized layout and the original layout (in term of fields); (4) a GIMPLE statement to assign the after-randomized field (or its location) to a temporary variable (or a pointer).

**Example.** We present an example in Figure 4 to illustrate how the randomization component works. Figure 4(a) shows the source code of the program; Figure 4(b) shows the original GIMPLE representation; Figure 4(c) shows the GIMPLE representation generated by SALADS. GIMPLE is a three-address representation in static single assignment form [4]. In a GIMPLE representation, temporary variables are defined to store the intermediate values for complex expressions. For example, in Figure 4(c), to allocate memory for the instance pointed by `p`, `D.2052` is temporarily defined to store the return value of `malloc` (line 15) and afterwards assigned to `p` (line 16). In particular, we explain how GIMPLE Pass-1 instruments the statement `p->a=1`. Suppose the definition of data structure TEST is identified. First, a GIMPLE statement to invoke Initialize_Record is inserted (line 18 Figure 4(c)). Initialize_Record initializes $I_a$ as `p` and $I_c$ as 0. Also Initialize_Record initializes $F_s$, $F_o$, and $F_f$ for each field in `p`. $F_s$ and $F_o$ are determined by the definition of TEST (line 1-7 Figure 4(a)); $F_r$ is set as the same with $F_o$; $F_f$ is set as 0 (i.e., randomizable). Second, a GIMPLE statement is inserted to call Update_Record (line 19 Figure 4(c)). Update_Record updates $I_c$ to be 1 and $I_a$ to be `p` and uses a routine `Shuffle(p)` to shuffle the layout, which are presented as step-1 to step-3 in Figure 3. The results are shown in Figure 3 after step-4. Third, a GIMPLE statement is inserted to call Offset_Diff (line 20 Figure 4(c)). Offset_Diff calculates difference between the after-randomization offset and original offset (presented as step-5 in Figure 3). For instance, the offset difference for `a` in `p` is 8. Fourth, a GIMPLE statement is inserted to assign the location of the randomized field to a pointer D.2058 (line 22 Figure 4(c)). Finally, the original statement `p->a=1` (line 9 Figure 4(b)) is replaced with a new statement `*D.2058=1` (line 23 Figure 4(c)).

### 3.3 De-randomization Component

Data structure randomization may introduce runtime errors. For example, a randomized data structure passed to an un-instrumented library function will be accessed based on the original layout. It will cause program errors because the function may access the irrelevant field in the randomized data structure.

The de-randomization component (i.e., GIMPLE Pass-2) is designed to avoid such errors. First, the pass scans the GIMPLE representation of a program to identify *dangerous statements*. A dangerous statement involves operations on randomized data structures and such operations might cause consequent inconsistency or crash. Second, the pass inserts a statement to invoke the de-randomization routine before a dangerous statement. This routine will restore the data structures involved in the dangerous statement into their original layouts. The dangerous statements appear in two scenarios as follows.

**Pointer involved dangerous statements.** There are two types of pointer-involved danger statements: (1) statements that cast a randomized data structure instance (or a randomized data structure pointer) `X` to another pointer `Y`, but `X` and `Y` are of different types. Such a statement is dangerous because the subsequent point-to-member operators over `Y` still access fields according to the original layout; (2) statements that use a pointer to reference a field in a data structure. Suppose there is a statement `int *p=&z.a`. When `z` is re-randomized after the assignment, the DSSR program cannot inform `p`. Consequently, `p` will point to an irrelevant field instead of `a`.

For the first type, the inserted de-randomization routine restores `X` to its original layout and mark it as un-randomizable. For the second type, the de-randomization routine restores the fields (e.g., `a`) referenced by pointers (e.g., `p`) to their original locations. Also, the routine marks such fields as un-randomizable.

**External and shared APIs involved dangerous statements.** Statements invoking external and shared APIs are dangerous if they pass data structure instances as arguments. For example, when a program calls `bind` in GNU LIBC with an instance of data structure `sockaddr`, the `sockaddr` instance might be randomized. However, `bind` still uses the `sockaddr` instance based on its original data structure layout. This will obviously lead to an execution error.

For such an API invoking statement, the inserted de-randomization routine will restore the data structure instances that are passed as arguments to their original layouts and mark them as un-randomizable.

### 3.4 Other Practical Issues

When there is a deep copy (e.g., plain assignment and `memcpy`) from data structure instance `A` to another instance `B` (`A` and `B` are with the same type), `B` shares the identical randomized layout with `A`. In our design, we directly copy the randomization record of `A` to `B`, except $I_a$ and $I_c$.

If multiple threads access the same data structure instance, the seed of the instance might turn into an un-synchronized state. For user space programs, we

leverage `pthread_mutext_lock` and `pthread_mutex_unlock` to keep the execution correct. For kernel space software, the DSSR programs rely on the spinlock interface `spin_lock` and `spin_unlock` to enforce synchronization.

A program might set a written protection attribute for pages that contain data structure instances. If so, DSSR programs firstly change attributes of these pages to make them writable and then randomize layouts of the instances.

## 4 Evaluation

We implement SALADS on top of gcc-4.5.0 with 11K lines of C code added. All evaluation experiments are conducted on an Intel(R) Core(TM) i5 machine with 4GB memory running Fedora Core Release 8 with Linux kernel version 2.6.23.1. In this section, we present the evaluation of the effectiveness and the performance of SALADS system.

### 4.1 Effectiveness of DSSR Application Programs

**How DSSR applications are generated.** We generate DSSR applications via using SALADS to compile open source programs, including SPECInt2000 [3], `httpd-2.0.6`, `openssh-2.1.1p4`, and `openssl-0.9.6d`. In principle, we should select security-sensitive data structures to randomize. However, we have limited knowledge about such data structures. To be general, we randomly select 20% of data structures to randomize in each program. In particular, determined security-related data structures are manually added to the randomization set.

**How attacks are launched.** We launch two real world attacks. In the first attack, we exploit the buffer overflow over the array `key_arg` in a data structure instance `session` (of type `ssl_session_st`) in `openssl` [15]. During the exploitation, the attack firstly overwrites the `key_arg` array and injects the shell codes. Then, the attack uses the pointer field `ciphers` in `session` to calculate the address of the shell codes. By substracting 368 from the pointer `session->ciphers`, the attacker can get the starting address of the shell code. Finally, the attacker redirects the program counter to the shell code. In the second attack, we exploit the integer truncation bug in [14], details of which have been presented before.

We also mimic a memory content leakage attack in the experiment: we insert a routine in each of the tested programs. The routine does two things. First, the routine dumps the page that contains the target data structure instance, immediately after the program receives inputs (e.g. socket packets). Second, the routine analyzes the dumped page to locate the base address of the target instance, based on the signature of the data structure. Signature of `passwd` in `openssh` has been explained previously. For `ssl_session_st`, the signature consists of 23 special fields (4 character arrays with 4 corresponding integer lengths, 6 pointer values, and 9 integer values). In addition, the routine can identify fields with unique features: `pw_uid` in `pw` is a small integer $\leq$ `0xFFFF`;

**Table 1.** Defense results of DSSR applications

| Programs | CVE # | Bugs | Data Structure | ASLR and DSLR | SALADS |
|---|---|---|---|---|---|
| openssl-0.9.6d | CVE-2002-0656 | KEY ARG bugs [15] | ssl_session_st | × | √ |
| openssh-2.1.1 | CVE-2001-0144 | CRC-32 bug [14] | passwd | × | √ |

`key_arg` in `session` is an 8-byte array which would very likely be separated from other fields by small values ($\leq$ `0x18`).

**Effectiveness of DSSR.** We compile the selected programs with static DSLR and SALADS, respectively. During our experiment, we also enable ASLR in the execution environments. We launch the two attacks to both static DSLR and SALADS compiled applications. Defense results are shown in Table 1. The results demonstrate that when memory content leakage happens, both ASLR and static DSLR cannot defend data structure manipulation attacks. However, SALADS is robust enough to prevent such attacks.

**Looking into the details.** Here we discuss the details of how SALADS defeats the two attacks. In the attack against `openssh`, the memory content leakage enables the attacker to infer the base address of `pw` and offset of `pw_uid` at the moment when the leakage happens. The attacker then manipulates the field `pw_uid` based on the inferred offset. However, a malicious request will trigger at least 5 accesses to `pw` before it overflows the target instance. Thus the target instance is re-randomized before being manipulated. The story is similar for the attack against `openssl`: a malicious request will trigger at least 17 accesses to `session` before it overwrites `key_arg`.

### 4.2 Effectiveness of DSSR kernel and DSSR hypervisor

**How DSSR Linux kernel and hypervisor are generated.** Linux kernel-2.6.23.1 contains 11430 data structure definitions. Randomizing all data structures would cause unacceptable overhead. In addition, we observe that many data structures are security in-sensitive and thus, should not be randomized. So we manually select 23 security-sensitive data structures (often used by the rootkits) from Linux kernel-2.6.23.1.

Xen-3.2.0 with Linux kernel-2.6.18.8 contains 11983 data structure definitions. We select 20 data structures from Xen-3.2.0 to randomize, which are widely used in security-sensitive source code files (e.g., mm.c). With the selected data structures, we compile the Linux kernel-2.6.23.1 and Xen-3.2.0 with SALADS.

**How attacks are launched.** We launch 12 widely used rootkits, as shown in Table 2, in the DSSR Linux kernel. These rootkits manipulate three data structures: `task_struct`, `proc_dir_entry`, and `module`. We also launch a Blue Pill attack against Xen-3.2.0, which reads and then manipulates the `vcpu` data structure with ring0 privilege. All the launched rootkits can circumvent OS level ASLR. The rootkits circumvent static DSLR in a similar way as explained before: speculate the layout of the target instance by referring another known instance of the same type (e.g., `proc_root` is a global variable of type `proc_dir_entry`).

**Table 2.** Defense results of deploying DSSR kernel against rootkits

| Rootkit Name | Data Structure | Description | prevented? |
|---|---|---|---|
| hideprocess-2.6 | task_struct | hide one process with given PID | √ |
| kbdy-2.6 | proc_dir_entry | privilege escalation to the user when open proc file | √ |
| adore-ng-0.56 | task_struct, proc_dir_entry, module | hide one process when open proc file | √ |
| taskigt | task_struct,proc_dir_entry | privilege escalation to process when open proc file | √ |
| enyelkm-1.3 | proc_dir_entry, module | hide module by modifying the proc read system call | √ |
| int3hook | module | hide process when hijacking int 3 | √ |
| synapsys | task_struct,module | give the root privilege to certain proess | √ |
| cleaner-2.6 | module | hide the next module of the rootkit | √ |
| linuxfu-2.6 | task_struct | hide the process given its name | √ |
| modhide | module | hide the module given its name | √ |
| override | task_struct | hide one process using injected code | √ |
| rmroots | task_struct, module | destroy static data structures to hide | √ |

**Effectiveness of DSSR.** We compile Linux kernel-2.6.23.1 and Xen-3.2.0 with static DSLR and SALADS, respectively. First, we execute the selected rootkits in the static DSLR kernel. These rootkits are enabled to infer the layout the target instance. The effects caused by these rootkits are presented in column 3 of Table 2 (titled as "Description"). Second, we execute the selected rootkits in DSSR kernel and enable them to infer the randomized layout as well. The experiments show two types of results: (1) the rootkit attack is prevented and the kernel continues to work without problems (`hideprocess`, `synapsys`, `linuxfu-2.6` and `override`); (2) the rootkit attack causes a kernel panic (the rootkit writes to a pointer which does not point to the location expected by the rootkit). Third, we launch the Blue Pill attack against static DSLR Xen-3.2.0 and DSSR Xen-3.2.0 and enable it to infer the randomized layout. Experiments show that static DSLR Xen is attacked but DSSR Xen is protected.

**Looking into the details.** Compared with user space programs, the kernel and the hypervisor contains many more data structure pointers. However, the SALADS system conducts de-randomization for many pointer involved operations. One potential issue is that many instances are de-randomized. For Linux kernel, we calculate the fields randomization rate (the percentage of randomizable fields in all fields) and instance randomization rate (the percentage of randomizable instances in all instances) during booting. In Table 3, we present the results for 17 data structures that are correlated to more operations than others. Field randomization rate for these data structures is 82.2% on average and instance randomization rate for these data structures is 80.9% on average.

### 4.3 Performance overhead

**Influence of threshold $W_m$ on performance overhead.** In SALADS, we set up a threshold $W_m$ to control the times of accesses between two successive randomization. In the first experiment, we use SPECInt2000 benchmark[5] to test how $W_m$ specifically affects the performance overhead introduced by SALADS. All data structures in these programs are randomized. $W_m$ is set to vary from 1 to 10 and for each value, we measure the average performance overhead. The

[5] We excluded three programs gcc, vortex, and eon in SPECInt2000 since these programs cannot be compiled with gcc-4.5.0.

**Table 3.** Randomization Rate of Data structure in Linux kernel-2.6.23.1 (Size: the memory size of the data structure (bytes); # Operations: the total DSSR statements inserted to handle operations on the data structure; $F_t$: the total number of fields in the data structure; $F_r$: the number of fields that can be randomized; $I_t$: the number of instances that are used; $I_r$: the number of randomized instances).

| Num | Name | Size | # Operations | $F_t$ | $F_r$ | $\gamma = \frac{F_r}{F_t}$ (%) | $I_t$ | $I_r$ | $\delta = \frac{I_r}{I_t}$(%) |
|---|---|---|---|---|---|---|---|---|---|
| 1 | sk_buff | 180 | 24770 | 44 | 42 | 95.5 | 1285 | 1086 | 84.5 |
| 2 | net_device | 1280 | 19918 | 100 | 91 | 91.0 | 76 | 72 | 96.1 |
| 3 | list_head | 8 | 15595 | 2 | 2 | 100 | 197391 | 160347 | 81.2 |
| 4 | task_struct | 1552 | 14171 | 130 | 97 | 74.6 | 386 | 386 | 100 |
| 5 | inode | 336 | 13779 | 44 | 25 | 56.8 | 5318 | 4772 | 89.7 |
| 6 | device | 328 | 12425 | 29 | 18 | 62.1 | 393 | 343 | 87.3 |
| 7 | super_block | 384 | 8970 | 38 | 25 | 65.8 | 23 | 21 | 91.3 |
| 8 | pci_dev | 996 | 7662 | 43 | 41 | 95.3 | 37 | 29 | 78.4 |
| 9 | socket | 40 | 5111 | 8 | 8 | 100 | 450 | 398 | 88.4 |
| 10 | Scsi_Host | 700 | 5050 | 59 | 56 | 94.9 | 40 | 34 | 85.0 |
| 11 | dentry | 132 | 4473 | 18 | 11 | 61.1 | 5408 | 5120 | 94.6 |
| 12 | urb | 104 | 4452 | 25 | 18 | 72.0 | 17 | 17 | 100 |
| 13 | scsi_cmnd | 304 | 4401 | 29 | 26 | 89.7 | 72 | 66 | 91.7 |
| 14 | buffer_head | 56 | 4226 | 12 | 9 | 75.0 | 8052 | 5155 | 64.0 |
| 15 | file | 132 | 4206 | 17 | 10 | 58.8 | 3436 | 2056 | 59.8 |
| 16 | net_device_stats | 92 | 4105 | 23 | 21 | 91.3 | 9 | 7 | 77.7 |
| 17 | sock | 364 | 3846 | 56 | 46 | 82.1 | 764 | 587 | 76.8 |

normalized results are shown in Figure 5. It can be observed that the performance overhead decreases as $W_m$ increases. When $W_m$ grows from 4 to 5, the performance overhead reduces sharply and after that, the performance overhead does not drop obviously. So we set $W_m$ to be 5 by default. All the following experiments are done with $W_m = 5$.

To evaluate the performance overhead introduced by SALADS, we test a variety of programs, including SPECInt2000, `httpd-2.0.6`, `openssh-2.1.1p4` and `openssl-0.9.6d`, Linux kernel 2.6.23.1 and Xen-3.2.0.

To evaluate user space applications, for testing SPECInt2000, we leverage the official benchmark; for testing `httpd`, we use apache benchmark; for testing `opensshl`, we use `openssl speed` [2]; for testing `openssh`, we upload 1.5GB test-files using `scp` [2] within 1000 times. The evaluation results are shown in Figure 6. The performance overhead introduced by SALADS ranges from from 0.2% to 23.5% on average. SALADS introduces higher performance overhead in `gzip`, `gap` and `twolf`. We find that the three programs leverage plenty of data structures to encapsulate data objects (e.g. compressed data, interpret dictionary word, and simulate objects) and frequently operate on these data structures. Consequently, DSSR statements are continuously executed in the three programs, which would cause high performance overhead.

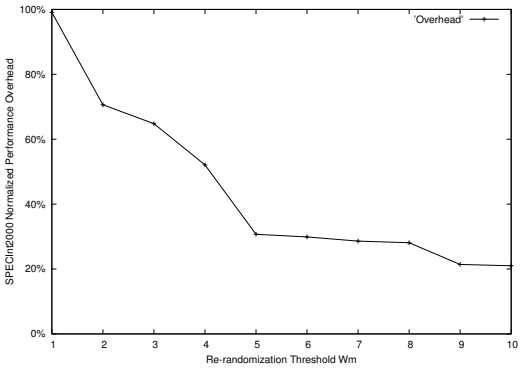



**Fig. 5.** Effects of $W_m$ on performance

For DSSR Linux kernel and DSSR Xen-3.2.0, we use the Lmbench [26] to evaluate the performance overhead. Specifically, we measure the overhead with

the bandwidth and the latency benchmarks. By only randomizing the selected data structures, DSSR Linux kernel introduces 6.7% to 28.8% (16.7% on average) runtime overhead, and DSSR Xen-3.2.0 introduces 0.1% to 14.8% (4.5% on average) runtime overhead. Details are presented in Table 4 in Appendix.

### 4.4 Memory Overhead

We measure the physical memories used by a set of DSSR programs and the corresponding original programs, at randomly selected time points during 1000 runs. As shown in Figure 7, the memory overhead introduced by SALADS to DSSR programs ranged from 0.7% (`openssh-2.1.1p4`) to 6.1% (`twolf`) on average. To measure memory overhead in DSSR kernels, we use the `dmesg` to get the memory usage of the Linux kernel after it is loaded. Both of the original Linux kernel and the DSSR Linux kernel are booted for three times to get the average memory usage. As shown in Figure 7, the DSSR Linux kernel introduces 8.6% memory overhead on average. We use the same method for Linux kernel to measure the memory overhead introduced by DSSR Xen-3.2.0. As shown in Figure 7, DSSR Xen-3.2.0 introduces memory overhead by 4.2% on average.

## 5 Discussion

### 5.1 Analysis of Effectiveness

Our threat model describes a simplified version of data structure manipulation attacks, which only involves one data structure and one specific field. In practice, a data structure manipulation attack often involves multiple data structures and multiple fields. For instance, the rootkit `taskigt` needs to read/manipulate `uid`, `gid`, `euid`, `egid` in `task_struct` and `read_proc` in `proc_dir_entry`, for a successful attack.

Here we discuss the difficulty introduced by SALADS to a data structure manipulation attack (suppose the original ASLR [32] is deployed). For generality, we assume (1) the attack needs to explore `n` data structure instances and the $i^{th}$ instance contains $l_i$ fields; (2) the attack needs to read/write $m_i$ fields in the $i^{th}$ instance; (3) the attack attempts to bypass the diversification defenses by brute force; (4) accesses to these instances are completed via one request. Attacks with multiple requests are separated into different attacks.

First, if the attack is against an application and no memory content leakage happens, the attack needs to crack both ASLR and SALADS. To bypass ASLR to refer the base addresses of the `n` data structures, the attack needs to make at most $2^{19} \times 3$ probes in total, because (1) the data structures may exist in randomized segment of `heap`, `stack`, or `data`; (2) a correct guess of one data structure in a single segment will reveal all other data structures in the same segment. Suppose ASLR has been bypassed and the base addresses of all the instances have been identified. To bypass SALADS to manipulate the correct fields, the attack needs to conduct $\prod_{i=1}^{n} \binom{l_i}{m_i}$ probes. Such a conclusion is based

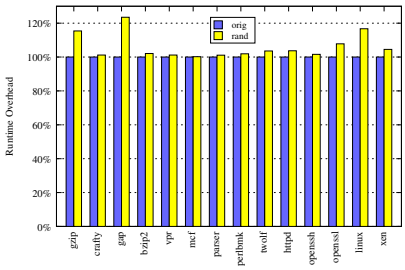



**Fig. 6.** Runtime Overhead

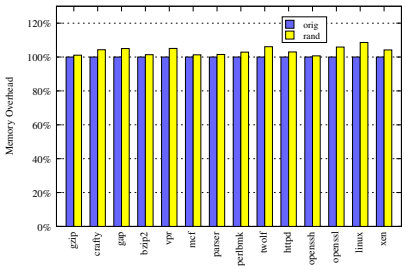



**Fig. 7.** Memory Overhead

on following facts: (1) all target fields in one single instance are to be accessed in one request, so the attack needs to guess all $m_i$ target fields in the $i_{th}$ instance in one probe; (2) all the $n$ instances are to be accessed in one request, which should be probed in one attempt. Summarily, SALADS complements ASLR to complicate data structure manipulation attacks. For instance, when $n = 2$, $l_1 = 19$, $l_2 = 130$, $m_1 = 1$, and $m_2 = 4$, the expected number of probes to crack SALADS is more than $2^{27}$ (the values are based on the `taskigt` rootkit).

Second, if the attack is against an application with memory content leakage or conducted by a rootkit, ASLR (and the static DSLR) is not effective. However, SALADS still works, which has been explained previously. Similarly, the attack needs to make $\prod_{i=1}^{n} \binom{l_i}{m_i}$ probes to bypass SALADS.

### 5.2 Limitations

In this section, we discuss the limitations of SALADS. First, our design does not explicitly protect the randomization records. Suppose an attacker can read arbitrary memories, including the randomization records. With the records, the attacker can recover the randomized layout. This is a common problem for compiler based defenses, such as Stackguard [13], and G-Free [27]. However, different from existing works, the leaked seeds might be invalid when the attack uses it. The time costs by memory content leakage varies from seconds [33] to weeks, when fine-grained ASLR protection is deployed. Within such a time window, a DSSR program might update the record for multiple times. To protect the randomization records, one possible solution is to adopt the key protection method proposed by Harrison [19]. This technique suggests introducing access control to prevent external code from accessing the key.

Second, an attacker may leverage code-reuse techniques to bypass SALADS. For example, the attacker could reuse the routines (e.g., `Update_Record`) to get the memory layout of a data structure. Fortunately, code reuse can be effectively handled by existing techniques, such as fine-grained ASLR for instruction areas [23,28,38,20,17] and control flow integrity (CFI) enforcement [5,37,41,42,16].

Third, it is hard to handle the balance between security and efficiency. To obtain the strongest protection, we should randomize all data structures, which, however, would introduce high performance overhead. In our current implementation, we randomize a subset of all data structures, including security-sensitive data structures. Common security-sensitive data structures include those contains authentication information or function pointers. To handle this limitation,

we provide users with a white list which contains data structures to be randomized. The users can add security-sensitive data structures into this list. In the near future, we plan to improve our current implementation to randomize more data structures and reduce the overhead.

## 6 Related Work

Plenty of techniques have been proposed to achieve address space randomization (ASR). These techniques introduce diversification to programs at different granularity [24], including segment level [36,9], page level [6], function level [23], basic block level [28], instruction level [38,20], and memory objects level [18,25,34,40].

In particular, Giuffrida et al. [18] propose a fine-grained OS-level live randomization technique, including data structure randomization. However, it has several limitations. First, their technique needs to heavily modify the microkernel-based OS; our technique can be directly applied to the targets with light-weight instrumentation. Second, their technique requires to separate a kernel into isolated components, which violates the design principles of modern kernels. Third, it cannot achieve live randomization in the microkernel; whereas our technique can be generically applied to applications, OS kernel and hypervisor code.

Static DSLR [25,34] is proposed to prevent data structure manipulation attacks, via modifying the definition of a data structure to reorder the fields. However, static DSLR has several limitations. First, the layout randomized by static DSLR is determined at compilation time. Second, static DSLR requires manual efforts to determine which data structure can be randomized. Xin et al. [40] extend static DSLR and propose to use a constraint set to select randomizable data structures. But their technique cannot handle nested data structures and ignores all data structures associated with pointer operations.

## 7 Conclusion

In this paper, we present SALADS, a system that automatically translates a program to a DSSR program. At runtime, a DSSR program adaptively randomizes the layout of each security-sensitive data structure independently. The randomizability of a data structure instance is automatically determined by the DSSR program. Experiments demonstrate both high effectiveness and reasonable performance when applying SALADS to defense against data structure manipulation attacks. As a technique to introduce artificial diversification, SALADS is robust enough to protect programs when memory content leakage happens and practically applicable to protect OS kernels and hypervisors against rootkits.

## 8 Acknowledgement

This work was supported in part by ARO W911NF-13-1-0421 (MURI), NSF CCF-1320605, and NSF CNS-1422594, Chinese National Natural Science Foundation (NSFC 61073027, NSFC 61272078).

## References


1. Arrays of length of zero. http://gcc.gnu.org/onlinedocs/gcc/Zero-length.html.
2. Openssh benchmark. http://blog.famzah.net/2010/06/11/openssh-ciphers-performance-benchmark/.
3. Spec cpu benchmark suite(2000). http://www.spec.org/cpu2000/.
4. Gimple, 2015. https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html.
5. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS '05)*, 2005.
6. M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium (Security '14)*, 2014.
7. A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Annual Computer Security Applications Conference (ACSAC '08)*, 2008.
8. S. L. BERRE. Bypassing windows 7 kernel aslr. 2011. http://dl.packetstormsecurity.net/papers/bypass/NES-BypassWin7KernelAslr.pdf.
9. E. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium (Security '03)*, 2003.
10. A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In *IEEE Symposium on Security and Privacy (Oakland '14)*, 2014.
11. E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh. Openconflict: Preventing real time map hacks in online games. In *IEEE Security and Privacy (Oakland '11)*, 2011.
12. H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Asia-Pacific Workshop on Systems (APSys '11)*, 2011.
13. C. Crispin, P. Calton, M. Dave, H. Heather, W. Jonathan, B. Peat, B. Steve, G. Aaron, W. Perry, and Z. Qian. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium (Security '98)*, 1998.
14. CVE-2001-0144. Ssh crc-32 compensation attack detector. 2001. http://www.securityfocus.com/bid/2347/discuss.
15. CVE-2002-0656. Apache openssl heap overflow exploit. 2002. http://www.phreedom.org/research/exploits/apache-openssl/.
16. L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nrnberger, and A.-R. Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones,. In *Annual Network and Distributed System Security Symposium (NDSS'12)*, 2012.
17. L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *Annual Network and Distributed System Security Symposium (NDSS '15)*, 2015.
18. C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Conference on Security Symposium (Security '12)*, 2012.
19. K. Harrison and S. Xu. Protecting cryptographic keys from memory disclosure attacks. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, 2007.

20. J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson. Ilr: Where'd my gadgets go? In *IEEE Symposium on Security and Privacy (Oakland '12)*, 2012.
21. R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium (Security '09)*, 2009.
22. R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *IEEE Symposium on Security and Privacy (Oakland '13)*, 2013.
23. C. Kil, J. Jim, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Annual Computer Security Applications Conference (ACSAC '06)*, 2006.
24. P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *IEEE Symposium on Security and Privacy (Oakland '14)*, 2014.
25. Z. Lin, R. D. Riley, and D. Xu. Polymorphing software by randomizing data structure layout. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '09)*, 2009.
26. L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *USENIX Security Symposium (Security '96)*, 1996.
27. K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Annual Computer Security Applications Conference (ACSAC '10)*, 2010.
28. V. Pappas, M. Polychronakis, and A. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy (Oakland '12)*, 2012.
29. Parvez. Bypassing microsoft windows aslr with a little help by ms-help. 2012. `http://www.greyhathacker.net/?p=585`.
30. J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures:remote side channel attacks on diversified code. In *ACM Conference on Computer and Communications Security (CCS '14)*, 2014.
31. F. J. Sern. Cve-2012-0769, the case of the perfect info leak. 2012. `http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf`.
32. H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM conference on Computer and communications security (CCS '04)*, 2004.
33. K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy (Oakland '13)*, 2013.
34. D. M. Stanley, D. Xu, and E. H. Spafford. Improved kernel security through memory layout randomization. In *International Performance Computing and Communications Conference (IPCCC '13)*, 2013.
35. R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *European Workshop on System Security (EUROSEC '09)*, 2009.
36. P. Team. Pax address space layout randomization (aslr). 2003. `http://pax.grsecurity.net/docs/aslr.txt`.
37. Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy (Oakland '10)*, 2010.
38. R. Wartell, V. Mohan, K. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM Conference on Computer and Communications Security (CCS '12)*, 2012.

39. Wikipedia. Data structure alignment. 2014. `http://en.wikipedia.org/wiki/Data_structure_alignment`.
40. Z. Xin, H. Chen, H. Han, B. Mao, and L. Xie. Misleading malware similarities analysis by automatic data structure obfuscation. In *International Conference on Information security (ISC '10)*, 2010.
41. C. Zhang, T. Wei, Z. Chen, L. Duan, S. McCamant, L. Szekeres, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy (Oakland '13)*, 2013.
42. M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *USENIX Security Symposium (Security '13)*, 2013.
43. Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *ACM Conference on Computer and Communications Security (CCS '12)*, 2012.

# A Details of Lmbench Results

**Table 4.** Lmbench results

| Latency | Linux kernel-2.6.23.1 | | | Linux kernel-2.6.18.8-xen0 | | |
|---|---|---|---|---|---|---|
| | orig (ms) | r(ms) | O(%) | orig (ms) | r(ms) | O(%) |
| Simple syscall | 0.1559 | 0.1791 | 14.9 | 0.2920 | 0.2949 | 1.0 |
| Simple read | 0.2239 | 0.2864 | 27.9 | 0.4021 | 0. 4082 | 1.5 |
| Simple write | 0.1972 | 0.2539 | 28.8 | 0.3658 | 0.3699 | 1.1 |
| Simple open/close | 1.8732 | 2.3089 | 23.2 | 2.7081 | 2.7323 | 0.9 |
| Process fork+exit | 60.2857 | 70.3026 | 16.6 | 342.8125 | 386.5949 | 12.8 |
| Select on 10 fds | 0.4458 | 0.5069 | 13.7 | 0.6874 | 0.6884 | 0.1 |
| Select on 100 fds | 1.3019 | 1.5419 | 18.4 | 1.7559 | 1.7688 | 0.7 |
| Protection fault | 0.2289 | 0.2497 | 9.1 | 0.5997 | 0.6009 | 0.2 |
| Pipe | 5.4670 | 5.8371 | 6.7 | 14.2375 | 16.3477 | 14.8 |
| AU_UNIX sock stream | 5.9704 | 7.0496 | 18.1 | 13.1883 | 14.6609 | 11.2 |
| Bandwidth | orig (MB/s) | r (MB/s) | O (%) | orig (MB/s) | r (MB/s) | O (%) |
| File I/O | 44.71 | 38.18 | 17.1 | 19.30 | 18.21 | 6.0 |
| Mmap I/O | 7423.26 | 6671.53 | 11.2 | 3023.09 | 2896.42 | 4.4 |
| Mem rd | 8359.75 | 7523.35 | 11.1 | 3378.98 | 3245.22 | 4.1 |

Table 4 lists the detailed results of testing DSSR Linux kernel ($2^{nd}$-$4^{th}$ columns) and DSSR Xen-3.2.0 ($5^{th}$-$7^{th}$ columns) with `Lmbench`. In particularly, we evaluate the performance overhead introduced by SALADS with two metrics: system latency and bandwidth. For DSSR Linux kernel, file operations (e.g., open/close) have higher performance overhead. Based on our observations, this is possibly because the file-related data structures (e.g., `inode`) contains many nested definitions which require more DSSR statements to complete the randomization. For DSSR Xen, the randomization mainly affects the local communications (e.g., Pipe) and process-related operations (e.g., process fork). This is probably because more DSSR statements at these points will be executed to access the privileged system components (e.g., MMU, I/O peripherals) and cause traps into the VMM.