Polymorphing Software by Randomizing Data Structure Layout

Zhiqiang Lin, Ryan D. Riley, and Dongyan Xu

CERIAS and Department of Computer Science, Purdue University, USA
{zlin,rileyrd,dxu}@cs.purdue.edu

Abstract. This paper introduces a new software polymorphism technique that randomizes program data structure layout. This technique will generate different data structure layouts for a program and thus diversify the binary code compiled from the same program source code. This technique can mitigate attacks (e.g., kernel rootkit attacks) that require knowledge about data structure definitions. It is also able to disrupt the generation of data structure-based program signatures. We have implemented our data structure layout randomization technique in the open source compiler collection `gcc-4.2.4` and applied it to a number of programs. Our evaluation results show that our technique is able to achieve software binary diversity. We also apply the technique to one operating system data structure in order to foil a number of kernel rootkit attacks. Meanwhile, programs produced by the technique were analyzed by a state-of-the-art data structure inference system and it was demonstrated that reliance on data structure signatures alone may lead to false negatives in malware detection.

1 Introduction

A widely adopted methodology for implementing software is data abstraction, which involves the abstraction of data structures and enables programmers to isolate a data definition from its representation and operations. Software is implemented to access and process data structures. Software implementation, if not obfuscated, will expose certain data structure definitions as well as their layouts. This observation has been exploited recently in network protocol reverse engineering [11, 16, 20, 29, 30, 40].

Knowledge about data structure layout is often used by attackers. For example, a buffer overflow attack relies on the attacker knowing that the program buffer is adjacent to a function pointer or return address [22]. Kernel rootkits, especially those that manipulate kernel objects directly, require that the attacker know the layout of specific kernel objects in order to manipulate them. In network application penetration testing, if the attacker knows the structure of the protocol message, he can reduce the fuzz space and speed up the test [21, 39]. These attacks can be foiled if we can prevent attackers from obtaining an accurate data structure layout of the victim program.

Data struct layouts are also used as attack signatures in some defense techniques. For example, in protocol analysis, the data structure associated with

a protocol payload can be used to construct the exploit signature for runtime network intrusion detection. In malware analysis, it has been reported recently that data structure layout can be used to generate malware signatures [19].

Forrest et al. [23] has suggested that monoculture is one of the main reasons why computers are vulnerable to large-scale, reproductive attacks. As such, randomization can be introduced to increase the diversity of software. This strategy has been widely instantiated in existing work such as address space randomization (ASR) [8, 10, 38, 41], instruction set randomization (ISR) [6, 28], data randomization [12,17], and operating system interfaces randomization [13,27]. Given the success of existing randomization strategies, we propose another instantiation of software randomization: Data structure layout randomization (DSLR).

In this paper, we demonstrate that software can be diversified by DSLR. We propose an approach to instrument a compiler (as the compiler knows about a program's semantics) so that it will generate a different data structure layout each time the same source program is compiled. We instrument the compiler to scan the data structure definitions (e.g., `struct` and `class`) marked by the programmer as randomizable and then reorder their member fields and insert garbage fields. We note that DSLR is different from the software obfuscation techniques [15]. Those techniques are used in software protection and aim at making it harder to reverse engineer the data structure definitions in a *single* binary. On the other hand, DSLR makes it difficult to derive data structure signatures from *multiple* copies of the same software.

The benefit of DSLR to malware defense is two-fold: First, DSLR can mitigate attacks that rely on knowing the data structure layout of victim programs. Second, the feasibility (and simplicity) of DSLR suggests that malware signatures based on data structure layout may not always be effective when used alone for malware detection.

We have implemented our DSLR technique in an open source compiler collection, `gcc-4.2.4`, and applied it to a number of programs. The detailed design and implementation are presented in Section 3 and Section 4, respectively. Our evaluation results in Section 5 show that DSLR can achieve software binary diversity. DSLR can be used generate diverse kernel data structure definitions to mitigate a number of kernel rootkit attacks. Meanwhile, we demonstrate that DSLR introduces noise to a state-of-the-art data structure inference system when generating a program's data structure signature. Finally, DSLR imposes very low performance overhead on `gcc` and on the original, un-randomized program.

2 Technical Challenges

In this section, we examine two technical challenges in realizing DSLR: Which data structures to randomize and how to randomize them.

2.1 Randomizability of Data Structures

Data structure layout, at the binary level, is reflected by the offsets of the encapsulated object fields. The encapsulated objects include `struct`, `class`, and stack

variables declared in functions (as they are related to a particular stack frame and addressed by `EBP`). The first two types have been exploited to derive malware signatures [19]. We believe that a function's local variable layout can also be leveraged to compose signatures and thus we will also discuss randomizing them.

However, randomizing just any data structure will not work in general as manifested in the following examples: (1) If a data structure is used in network communication, the communicating parties may not understand each other if the data structure is randomized. (2) If a data structure definition is public (e.g., defined in shared library `stdio.h`), it cannot be randomized. (3) There is a special case in GNU C that allows zero-length arrays to be the last element of a structure (a zero-length array is actually the header of a variable-length object). If a zero-length array is declared as the last element in a `struct`, that element cannot be randomized, otherwise it cannot pass `gcc` syntax checking. (4) A programmer may directly use the data offset to access some fields. (This is particularly true in programs which mix assembly and C code.) (5) To initialize the value of a structure, the programmer uses the order declared to initialize the structure. These fields cannot be randomized, as the program may crash. In light of these cases, we declare a data structure as randomizable if and only if it is not exposed to any other external programs and does not violate the original `gcc` syntax and programmer intention.

Data structure randomizability is closely related to program semantics. It would be ideal if the compiler could automatically spot all the randomizable data structures. In practice, however, only the programmer can designate randomizable data structures with confidence. Even if we could define some heuristics to automatically spot those randomizable data structures, we could not claim both completeness and safety. In this paper, we simply require that programmers use new keywords to specify randomizable data structures.

2.2 Data Structure Randomization Methods

The second challenge is how to randomize a data structure. The simplest randomization method would be to reorder its layout. Our primary goal is to create binary diversity for the same software – the more variation, the better. Therefore, we will design a randomization method which reorders the member fields of each data structure to be randomized. Suppose a program has n such data structures and each has m fields, then the number of possible combinations after randomization would be $(m!)^n$.

However, field reordering alone is still not sufficient. For example, suppose a data structure has only two members which are both of `int` type. No matter how we reorder these two fields, the layout of this data structure is still "`int` and `int`". As a result, to randomize a data structure containing multiple members of the same type, we have to use a different randomization method. To this end, we insert garbage fields into these data structures.

3 DSLR Design in GCC

In this section we present the detailed design of DSLR in a specific compiler system. As C/C++ is commonly used in system and user level programming, we have implemented our DSLR technique in the popular, open-source compiler `gcc` [1].

By instrumenting `gcc` to reorganize the fields in encapsulated data structures, DSLR will fill the memory image with a random layout each time the program source is compiled. Hence, we need to decide where to instrument `gcc`.

For a program source, `gcc` first builds an initial Abstract Syntax Tree (AST). It then converts the language-specific AST into a uniform, generic AST. The generic AST will be transformed into another representation called GIMPLE (a representation form which has at most three operands). After GIMPLE, the source code is converted into the static single assignment (SSA) representation [5] to facilitate more than 20 different optimizations on SSA trees. After the SSA optimization pass, the tree is converted back to GIMPLE which is then used to generate a register-transfer language (RTL) tree. RTL is a hardware-based representation that corresponds to an abstract target architecture with an infinite number of registers. There are also a number of optimization passes such as register allocation, code scheduling, and peepholes performed at the RTL level.

Given these internal steps in `gcc`, the possible instrumentation points for DSLR are AST, GIMPLE, SSA, and RTL. We instrumented at the AST level for the following reasons: (1) the AST retains a lot of original information from the program source code, such as the type and scope information for data structures and functions; (2) The AST representation is easier to understand and the structure of the tree is concise and relatively convenient for us to modify; (3) When generating the AST, `gcc` has not yet determined the layout of the data structures, and as such we can reorder the data structure members and reconstruct the AST without needing to compute specific memory addresses.

The data structures to be randomized can be divided into three categories: `struct`, `class` and the function stack variables. We reorder the inner AST representations of these data structures, which will eventually lead to the reorganization of the memory layout. Note that these data structures have their own scopes. When the AST for these data structures is generated, all the member variables in each data structure are chained together and represented by a link list. To perform randomization, we can just capture the head node of the list, reorder the nodes of the list based on a random seed, and insert some "garbage" nodes into the list if necessary.

Figure 1 shows a simple example. A data structure `test` has three fields: `int a`, `char b`, and `int* c`. When compiled with the original `gcc`, the order of the fields is in the originally declared order (Figure 1(b).) When compiled with our DSLR-enabled `gcc`, the order of the fields is randomized. We also add 2 garbage fields. Figure 1(c) shows the randomized AST representation of `struct test`.

As discussed in Section 2, to enhance data structure layout diversity we adopt the following strategy: (1) different data structures at the same project build-

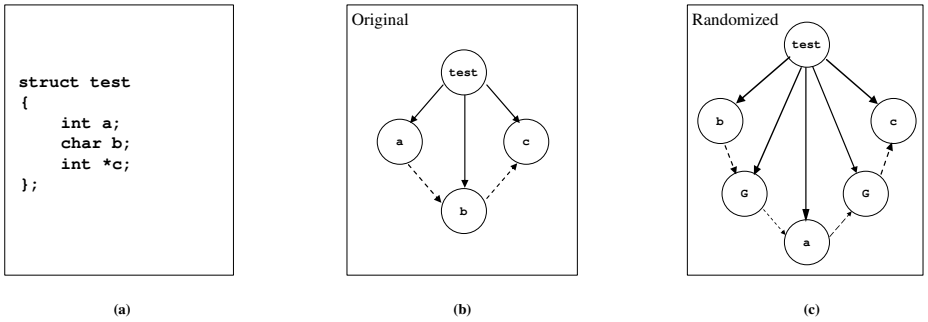

Fig. 1: Example of data structure randomization: (a) the original definition, (b) the original AST, and (c) the randomized AST. The "G" nodes represent the garbage fields added to the data structure. The dotted arrows represent the order of the fields.

ing time will be reordered differently (with different randomization seeds); and (2) the same data structure at different project building times will be reordered differently. We use project building time instead of compile time because when building a project, `gcc` usually compiles each file individually (as specified in the Makefile), and we need to ensure that the same data structure has a uniform layout across one entire build. Suppose a program has two data structures, `S1` and `S2`, which have 4 and 5 fields respectively. When we build the program using our modified `gcc`, `S1` and `S2` will be randomized differently. In addition, the same data structure (e.g., `S1`) will have different layouts in memory at different project building times. Hence, the number of possible layouts for this program would be $4! * 5!$. We believe such a strategy will greatly improve the binary diversity of the program, as the chances of generating identical instances would be $1/(\prod_{i=1}^{j} |S_i|!)$, where j is the total number of data structures to be randomized and $|S_i|$ represents the total number of fields (members) in data structure S_i.

4 DSLR Implementation in GCC

Our DSLR prototype is implemented in `gcc-4.2.4` with over one thousand lines of C code. We modified `gcc`'s AST representation to perform the randomization. Our prototype consists of four key components: (1) keyword recognizer, which recognizes the new keywords we introduce to specify data structure randomizability and garbage padding; (2) re-orderer, which reorders the field variables in a data structure definition according to a random seed; (3) padder, which inserts the garbage fields into a data structure; and (4) randomization driver, which controls the randomization process. In the remainder of this section, we present the details of these components.

4.1 Keyword Recognizer

We introduce several new keywords to instruct `gcc` regarding which data structures to randomize and how.

```
...
<function-definition>          ::= {<declaration-specifier>}*<declarator>{<declaration>}*<compound-statement>
<declaration-specifier>        ::= <storage-specifier>
                                 | <obfuscate-specifier>
                                 | <type-specifier>
                                 | <type-qualifier>
<obfuscate-specifier>          ::= __obfuscate__(( <obfuscate-list> ))
<obfuscate-list>               ::= <obfuscate-property>
                                 | <obfuscate-list>, <obfuscate-property>
<obfuscate-property>           ::= ε | __reorder__ | __garbage__
<struct-or-union-specifier>    ::= <struct-or-union> <identifier> "{" {<struct-declaration>}+ "}" <obfuscate-specifier>
                                 | <struct-or-union> "{" {<struct-declaration> <obfuscate-specifier>}+ "}"
                                 | <struct-or-union> <identifier> <obfuscate-specifier>
<class-specifier>              ::= <class> <identifier> "{" {<class-declaration>}+ "}" <obfuscate-specifier>
                                 | <class> "{" {<class-declaration> <obfuscate-specifier>}+ "}"
                                 | <class> <identifier> <obfuscate-specifier>
...
```

Fig. 2: A partial BNF definition of our extend grammar for C/C++.

The first keyword is `__obfuscate__`. It is implemented similar to the way `__attribute__` is already implemented in [3]. Similar to `__attribute__`, we offer options for `__obfuscate__` to tell `gcc` which randomization method(s) it should apply. For that we define two other keywords: `__reorder__` and `__garbage__`. The first one informs `gcc` that the data structure layout should be reordered and the latter one tells `gcc` to insert some garbage fields into the data structure.

There are three types of data structures that can be randomized and marked with the `__obfuscate__` keyword: (1) `struct`s in C, (2) `class`es in C++, and (3) stack variables declared in a function. Figure 3 shows usage examples of these keywords.

```
1 class Test
2 {
3    int a;
4    char b;
5    int *c;
6    ...
7 } __obfuscate__ (( __reorder__ ));
```

(a)

```
 1 #include <stdio.h>
 2 struct Test
 3 {
 4    int a;
 5    char b;
 6    int *c;
 7 } __obfuscate__ (( __reorder__ , __garbage__ ));
 8 __obfuscate__ (( __reorder__ )) int main(void)
 9 {
10    int loc1 = 1;
11    char loc2 = 'n';
12    char loc3[4];
13    printf(" The address in struct:
14             %x , %x , %x\n", &t.a, &t.b, &t.c);
15    printf(" The address in local:
16             %x, %x, %x\n",&loc1,&loc2,&loc3);
17    return 0;
18 }
```

(b)

Fig. 3: Sample code (a) showing how to randomize a class in C++ and (b) showing how to randomize a `struct` and stack variables in the `main` function.

Since we implemented DSLR at the AST level, there are two modifications when implementing the new keywords. The first is in lexical analysis, which makes the compiler recognize the new token. The second is to build our own *parser* for the keyword.

4.2 Reorderer

When generating the AST for a program, `gcc` will chain the members of a particular data structure to a list. If it encounters the keyword `__reorder__`, it will invoke the re-orderer when `gcc` finishes constructing the entire chain, and then it can reorder the members according to the random seed generated by the randomization driver.

We implement the re-orderer at different points for each category of data structures. To randomize the layout for a `struct`, we insert the re-orderer into function `c_parser_struct_or_union_specifier`, which handles `struct`s and `union`s, just after this function has constructed every item in a `struct` or `union`. Note that it is not necessary to randomize the members in a `union` as it only contains one instance of the declared members at runtime. To randomize a `class`, we insert our re-orderer into function `unreverse_member_declarations`. For local variables, we insert it into the function `c_parser_compound_statement_nostart`.

4.3 Padder

We implement the padder to insert garbage fields between fields of a data structure. The padder will be combined with the re-orderer to perform the randomization, and it will be inserted in the same places as the re-orderer. When `gcc` recognizes the keyword `__garbage__`, the padder will insert garbage fields of various sizes. Such garbage creates noise in the memory image and makes it more difficult to identify the true data structure. The size of garbage items is determined by the randomization driver.

4.4 Randomization Driver

The randomization driver supports the re-orderer and padder and is directly related to the effectiveness of DSLR. When encountering a randomizable data structure during project building, it will first check whether this data structure already has a 32-bit random value stored in a project build file. If so, it will use that random value; otherwise it will generate a random value via the `glibc` function `random` and store it in the project build file for future use. The project build file is a project-wide file that records the random value and the number of fields of each data structure to be randomized. It is critical to ensuring layout consistency across a single project build. In particular, when building projects such as the Linux kernel and its drivers, it should use the same project build file, otherwise the kernel may use different data structure layouts and cause crashes. Similarly, it checks whether the total number of elements of that data structure has been counted. If not, it will count the number of fields in that data structure and store it in the project build file.

After knowing the random value and the total number of fields for a data structure to be randomized, it takes two basic methods to perform the re-ordering and padding.

Reordering Method Suppose our randomization driver gets a random value R and the total number of fields for a particular data structure m. It will follow the reordering method shown in Algorithm 1.

Algorithm 1 *Reordering Method*

```
1: Input: random value R, total number of fields m, and the original order of field variables:
   pos[1..m]
2: Output: the reordered fields in pos'[1..m]
3: Initialization: j ← m;
4: Reorder(j, pos[1..j]){
5:     i ← R%j + 1;
6:     pos'[j] ← pos[i]; /*move the i-th element in pos to the rightmost available position in pos'*/
7:     if(j==1) return;    /*no element left in pos, and hence return*/
8:     if(i!=j) pos[i] ← pos[j];
9:     Reorder(j-1, pos[1..j-1]);
10: }
```

In the algorithm, `pos[i]` represents the position of the ith member/field variable in the original data structure. Based on the original ordering of the member variables, the method recalculates the positions of the member variables according to the random value R. We verify that Algorithm 1 is able to generate all $m!$ layouts for a data structure containing m members.

Padding Method When we insert garbage fields between the member variables of a data structure, the padding method determines the size of the garbage fields. We limit the size to 1, 2, 4, or 8 bytes. To do that we partition the random value R into four parts: x_1, x_2, x_3, and x_4, and each part has 8 bits. We then reduce these 8 bits to 2 bits by calculating x_i mod 4 ($i \in \{1, 2, 3, 4\}$). These four random values fall into the range of 0 to 3, which correspond to 8-byte, 4-byte, 2-byte, and 1-byte sizes, respectively. Suppose there exists a data structure which contains five member variables and the four random values (after the mod operation) are 1, 3, 2, and 0. Then we insert 4 garbage fields between the members using padding size of 4, 1, 2, and 8 bytes, respectively. Note that if the data structure requires both reordering and padding, the two methods will be applied in that order. We note that padding will not interfere with any subsequent optimization steps performed by `gcc`.

5 Evaluation

In this section, we present our evaluation results. We first assess the effectiveness of our DSLR technique in Section 5.1, and then measure the performance impact of DSLR on both `gcc` and the generated binaries in Section 5.2.

5.1 Effectiveness

Estimating Data Structure Randomizability We applied our DSLR-enabled `gcc` to a number of goodware and malware programs. We use open-source goodware such as openssh, and malware programs collected from offensive computing [2] and VX Heavens [4]. We first manually estimate the randomizability of

data structures in these programs by inspecting their source code. As discussed in Section 2, it is difficult to accurately determine all the randomizable data structures in a program and we delegate that task to programmers. In our experiments, we used the following heuristics for randomizability estimation: For each data structure, we manually check if it is used/involved in one of the following scenarios: (1) network communication, (2) disk I/O, (3) shared library, (4) assembly code, (5) pointer arithmetic, and (6) struct data initialization. If so, the data structure is deemed un-randomizable.

Table 1 summarizes the results. We define k_i ($i \in \{0, 1, 2\}$) as the total number of `struct`s, `class`es, or functions in a program. We also define j_i ($i \in \{0, 1, 2\}$) as the total number of data structures we consider randomizable. Hence, j_0/k_0, j_1/k_1, and j_2/k_2 represent the randomizability ratios for `struct`, `class`, and function (shown in the 3^{rd}, 4^{th}, and 5^{th} columns in Table 1), respectively. We note that some of the function stack layouts could not be randomized. The reason is that they contain `goto` statements (thus the label order is fixed).

Benchmark program	LOC(K)	Randomizability of Data Structure			Possible Layout
		struct	class	funcs	ω
42 Virus	0.88	1/1	-	24/24	4E5
Slapper	2.44	26/30	-	69/70	5E47
pingrootkit	4.81	26/27	-	57/57	5E15
Mood-nt	5.31	36/37	-	121/122	8E119
tnet-1.55	11.56	14/17	-	179/179	7E82
Suckit	24.71	110/111	-	143/144	9E159
agobot3-pre4	245.44	23/31	50/50	340/346	2E1106
patch-2.5.4	11.53	5/7	-	123/123	4E3
bc-1.06	14.29	20/21	-	166/166	6E56
tidy4aug00	15.95	9/18	-	341/341	2E52
ctags-5.7	27.22	51/79	-	488/488	3E668
openssh-4.3	76.05	63/80	-	820/838	4E1271

Table 1: Result of randomizability estimate and layout diversity

Layout Diversity `bc_struct` is a data structure in the `bc-1.06` binary. As shown in Figure 4, this data structure compiled by the DSLR-enabled compiler (with random number 669) has its layout changed significantly: not only has the field order been changed, it also contains 6 additional garbage fields.

We then estimate the layout diversity of these programs. It is rather cumbersome to experiment with all possible layouts. Instead, we numerically compute the number of binary variants that our compiler will be able to generate for each program, based on the result of the data structure randomizability estimation (j_i ($i \in \{0, 1, 2\}$) of each program). The numerical results are shown as ω in the last column of Table 1, which is the total number of binary instances of each program. Note $\omega = \prod_{i=1}^{j_0+j_1} |S_i|!$, where $j_0 + j_1$ is the total number of `struct`s and `class`es to be randomized and $|S_i|$ is the total number of fields (members) in data structure S_i.

Binary Code Diversity A direct consequence of randomizing data structure layout is that the binary code generated will also be diversified. The reason is that

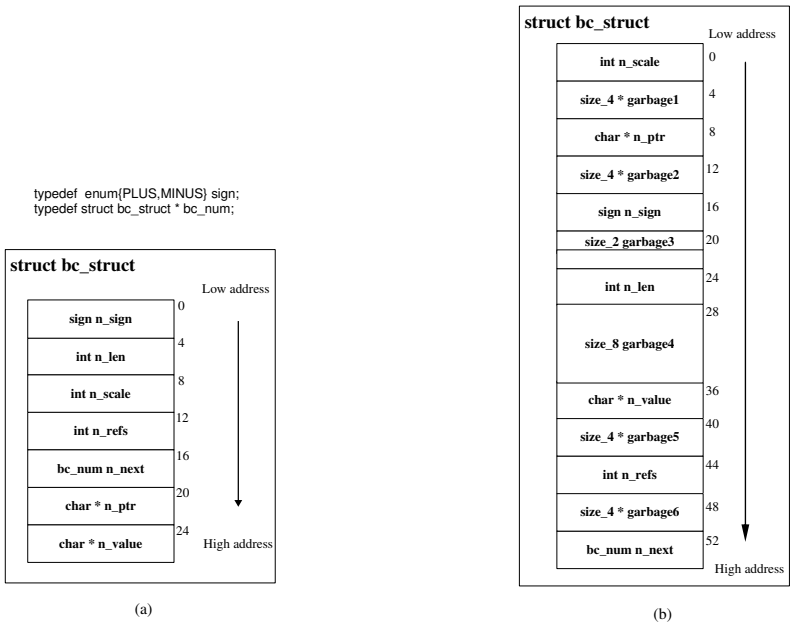

Fig. 4: Data structure layout comparison: (a) original layout, and (b) randomized layout

the field variables in `structs`, `classes`, and even local variables, are accessed by data offsets which will be changed due to the randomization. Therefore, it would be interesting to evaluate the difference between the DSLR-generated code and the original un-randomized code.

To evaluate code diversity, we first compiled each benchmark program with an unmodified copy of `gcc` to get the original binary, whose size is represented by I_0 shown in the 2^{nd} column of Table 2. We then used the DSLR-enabled `gcc` to compile the same program and generate three instances. Their code sizes are represented by I_1, I_2, and I_3, respectively. Next, we compared the original binary with the newly generated binaries using a tool called `bsdiff` [33]. The difference is represented by δ_i. `bsdiff` is a patch tool which generates the difference between two binaries. Different from other binary diff-ing tools, `bsdiff` adopts an "approximate matching" algorithm, which counts the byte-wise difference in two directions (both forward and backward) rather than in one direction (often forward). As such the results generated by `bsdiff` are more accurate. Note that the results of `bsdiff` are highly compact [33], and thus the differences reported by `bsdiff` are relatively small. According to `bsdiff`, DSLR can achieve a difference between 3-17%. The last column of Table 2 $\overline{AVG_\delta}$ shows the average percentage over the three instances.

Defending Against Kernel Rootkits A kernel rootkit is a piece of malicious software that compromises a running operating system kernel. Usually an at-

Benchmark program	Code Diversity							
	I_0(K)	I_1(K)	δ_1(%)	I_2(K)	δ_2(%)	I_3(K)	δ_3(%)	$\overline{AVG_\delta}$
42 Virus	27.37	27.39	7.0	27.39	6.5	27.40	8.0	7.2%
Slapper	36.03	33.83	12.0	33.85	13.2	33.82	14.3	13.2%
pingrootkit	84.08	84.29	5.0	84.28	3.9	84.28	5.1	4.7%
Mood-nt	74.52	75.25	9.6	75.32	9.5	75.35	9.8	9.6%
tnet-1.55	174.17	175.15	7.6	175.03	7.7	174.96	6.9	7.4%
Suckit	99.61	102.20	6.3	102.17	6.8	102.17	6.4	6.5%
agobot3-pre4	904.42	909.97	8.3	912.72	8.5	909.55	7.2	8.0%
patch-2.5.4	216.56	217.51	6.1	217.48	6.2	217.51	6.2	6.2%
bc-1.06	150.39	151.64	8.8	151.55	8.2	151.57	8.2	8.4%
tidy4aug00	119.54	119.54	6.7	119.54	6.8	119.54	7.5	7.0%
ctags-5.7	527.11	531.69	16.2	531.69	16.4	531.64	16.7	16.4%
openssh-4.3	997.64	1003.39	8.5	1003.53	8.2	1003.52	8.1	8.3%

Table 2: Evaluation of binary code diversity.

tacker will use them to hide his presence on a running system. An important feature of modern kernel rootkits is their ability to hide the existence of running processes from an administrator. It is important, for example, that malicious processes not appear in `ps` listings. To evaluate our DSLR-enabled compiler as a defense solution, we used it to randomize the `task_struct` data structure in the Linux kernel (version 2.6.8) to protect against these process hiding attacks by a number of kernel rootkits. Six rootkits were tested to determine if they were able to hide a process under the randomized kernel. A summary of the results is shown in Table 3. Detailed results for each rootkit are as follows:

adore-ng The adore-ng rootkit is a loadable kernel module (LKM) rootkit. This means that it is loaded into the kernel like a driver. After being loaded, adore-ng modifies function pointers contained in various kernel data structures. It avoids the system call table, as hooking the system call table would make it easily detectable. Adore-ng also has a user-level component, `ava`. When `ava` authenticates with the rootkit, a flag is added to the `flags` element of the `task_struct` for the `ava` process. Under the newly randomized kernel the `flags` element cannot be accurately located, and so `ava` cannot be properly authenticated. This renders the rootkit useless.

enyelkm While still being an LKM, enyelkm differs from adore-ng in that it does not have a user-level control component. Instead, options are chosen at compile time. By default, enyelkm hides any running process whose name contains the string OCULTAR. It finds these processes by traversing the process list and scanning the process names. Under the randomized kernel the linked list within `task_struct`s is randomly located, making enyelkm's attempts to traverse the list fail. This causes process hiding to be unsuccessful.

override Much like enyelkm, override is configured at compile time. Override makes extensive use of `current`, which is a macro that resolves to be the address of the `task_struct` for the currently running process. When running on the new kernel, the randomized elements of this data structure cause override to crash the kernel.

fuuld Fuuld is a data-only rootkit written by one of this paper's authors during previous research. It uses a technique known as direct kernel object manipulation (DKOM) to modify kernel objects directly without the need to execute code in the kernel. It operates by using `/dev/kmem` to search for and remove processes from the process list. When the `task_struct` structure is randomized, it is unable to properly traverse the process list.

intoxnia-ng2 The intoxnia rootkit is another LKM rootkit. Unlike adore-ng, however, intoxnia compromises the kernel by only hooking the system call table. Interestingly, this simplistic attack method is not troubled by the randomization of `task_struct`. This is because intoxnia hides a process by filtering the data returned by the system call `getdents` to ensure that directory listings from the `/proc` file system do not reflect hidden processes. Neither the process list, nor any elements in it, are involved. The data structures that intoxnia does modify are arguments to system calls, which cannot be randomized because they are part of the user-level library as well.

mood-nt The mood-nt rootkit installs itself directly into the running kernel using the `/dev/kmem` interface. It then proceeds to hook the system call table and hide processes using a technique similar to that of intoxnia. As such, this rootkit was also uninhibited by the randomization of `task_struct`.

Many kernel rootkits operate by inserting malicious code into the kernel and modifying existing function pointers to cause the kernel to execute it. Five of the above rootkits (adore-ng, enyelkm, override, intoxnia, and mood-nt) employ this attack strategy. Existing work [24, 35, 37] is able to effectively prevent these attacks. However, a different type of rootkit attacks – data-only attacks – exist. In this case, a rootkit program will directly modify kernel data structures using a memory interface device such as `/dev/kmem`. The fuuld rootkit above employs this strategy. As evidenced by its effectiveness against the fuuld rootkit, DSLR appears to be a promising approach to defending against data-only attacks. Given that the rootkit author must know the layout of kernel data structures in order to modify them, randomizing that layout will significantly raise the bar for such attacks.

Rootkit	Attack Vector	Prevented?
adore-ng 0.56	LKM	✓
enyelkm 1.2	LKM	✓
override	LKM	✓
fuuld	DKOM – `/dev/kmem`	✓
intoxnia ng2	LKM	×
mood-nt	`/dev/mem`	×

Table 3: Effectiveness of DSLR against kernel rootkits

Evaluation against Laika We also performed effectiveness evaluation of DSLR against Laika [19], a data structure inference system. The released version of Laika only supports taking snapshots of Windows binaries, whereas we implement DSLR in `gcc`, which cannot compile Windows programs. To assess the effectiveness of DSLR, we had to manually randomize the data structures in a Windows-based program by following our randomization methods. We then used

the Windows compiler to generate the binary code. We used three Windows-based programs: agobot, 7-zip, and notepad. For some reason, Laika could not process the binary image of notepad. Hence we only present the results with 7-zip and agobot.

For each application, we generated three binary instances and used Laika to detect their data structure layout similarity. In particular, Laika uses a mixture ratio [19] to quantify similarity: the closer the value is to 0.5, the higher the similarity. When detecting similarity, Laika has the option of filtering out pointers. Table 4 summarizes the results. The code difference among the instances of each program is around 5%. For 7-zip, when pointers are filtered out, Laika reported mixture ratios around 0.502. With pointers, it reported mixture ratios around 0.511. It looks like the binaries of 7-zip do not appear significantly different to Laika. We believe that the reason is the following: 7-zip only has 25 data structures randomized. But it has more than 80 un-randomizable data structures which are in the library. These data structures dominated. Hence the mixture ratios are close to 0.5. Agobot, on the other hand, contains 49 data structures and 50 classes in its own code, so the mixture ratios went higher: 0.57 without pointers and 0.63 with pointers. The mixture ratios indicate that, by randomizing the data structure layout, we introduced noise to Laika. Also, even though Laika indicated high similarity among 7-zip instances, it is still debatable how to account for the library code when detecting data structure similarity, as two different applications (with a small number of user-level data structures) may use lots of similar library data structures (such as those in the runtime support) in their implementations.

Benchmark Program	LOC	Un-randomized Binary	Randomized Binary	Code Difference	Mixture Ratio	
					w/o Pointer	w/ Pointer
7zip-4.64	41.01K	498K	502K	4.26%	0.50184625	0.50942826
			503K	5.08%	0.50244766	0.51070610
			504K	5.88%	0.50325966	0.51487480
agobot3-0.2.1-priv4	497.09K	1.17M	1.18M	6.18%	0.57368920	0.70016150
			1.19M	6.10%	0.57586336	0.60932887
			1.19M	6.34%	0.56068546	0.58418036

Table 4: Evaluation of DSLR against Laika

5.2 Performance Overhead

Finally, we evaluate the performance overhead incurred by DSLR. Since we modified `gcc`, we would like to know how much overhead DSLR imposes on `gcc`. In our experiments, we built each program 3 times. g_1, g_2, and g_3 represent the normalized `gcc` performance overhead. The 2^{nd}, 3^{rd}, and 4^{th} columns of Table 5 show these results. On average DSLR imposed around 2% performance overhead, which is mainly caused by random value lookup, field count, and field reordering.

Since DSLR will change the program's data structure layout and subsequently change the binary code produced, we would also like to know the program's performance overhead due to DSLR. We measured the corresponding

runtime overhead of the compiled binaries. The 6^{th}, 7^{th} and 8^{th} columns of Table 5 show these results. DSLR imposed less than 4% overhead. The normalized overhead is obtained by running each binary 10 times. Note that for those virus and daemon malware programs, we did not measure their performance overhead (i.e. the N/As in Table 5) as they ran in the background and were thus difficult to measure. We notice that some randomized binaries reported a slightly better performance than their un-randomized counterparts. The reason may lie in the data locality improvement caused by DSLR.

Benchmark program	Overhead imposed to `gcc`				Overhead imposed to application			
	g_1	g_2	g_3	$\overline{AVG_g}$	o_1	o_2	o_3	$\overline{AVG_o}$
42 Virus	3.6%	3.2%	2.7%	3.2%	N/A	N/A	N/A	N/A
Slapper	2.5%	2.8%	2.1%	2.5%	N/A	N/A	N/A	N/A
pingrootkit	3.0%	2.8%	2.7%	2.8%	N/A	N/A	N/A	N/A
Mood-nt	2.2%	2.1%	1.8%	2.0%	N/A	N/A	N/A	N/A
tnet-1.55	0.8%	1.2%	1.1%	1.0%	N/A	N/A	N/A	N/A
Suckit	1.2%	1.5%	2.3%	1.7%	N/A	N/A	N/A	N/A
agobot3-pre4	2.9%	3.3%	3.0%	3.1%	N/A	N/A	N/A	N/A
patch-2.5.4	1.6%	1.0%	1.2%	1.3%	-0.9%	1.2%	-2.0%	-0.6%
bc-1.06	3.0%	0.9%	2.4%	2.1%	1.1%	1.0%	-0.8%	0.4%
tidy4aug00	1.7%	1.5%	1.8%	1.7%	1.6%	-1.3%	1.1%	0.5%
ctags-5.7	2.9%	1.8%	1.1%	1.9%	-1.8%	-0.7%	-0.7%	-1.1%
openssh-4.3	1.7%	2.4%	1.8%	2.0%	2.7%	1.8%	-0.9%	1.2%

Table 5: Normalized performance overhead.

6 Limitations and Future Work

The first limitation of DSLR is that right now we do not support other languages such as Java, as we instrument `gcc` at the language-specific AST level. Our next step will involve either adding support to these languages, or studying the details of other `gcc` internal representations such as GIMPLE and RTL so that DSLR support can be made more generic.

The second limitation is that the randomizability of a data structure cannot be determined accurately and automatically. Instead, we rely on programmers' knowledge and judgment. As discussed in Section 2, the fundamental challenge in automatically determining the randomizability of a data structure is safety and completeness. To automate the identification, we could approximate the result by performing some sort of data flow analysis to identify certain un-randomizable data structures. For example, if we do not aim at achieving completeness, we could adopt several heuristics to achieve automation, such as using the execution context to determine if a data structure is used in network I/O and excluding it from DSLR if so.

The third limitation is that we do not support other randomization techniques such as struct/class splitting. Right now we only increase the field number by adding garbage fields, and we do not decrease the field numbers, which can be achieved by struct/class splitting techniques used in the obfuscation community [15]. Our future work includes adopting those obfuscation techniques to make it generate more polymorphic data structure layouts.

The fourth limitation is in software distribution. When compiled by the DSLR-enabled `gcc`, a program can have a large number of binary variants. It will cause some inconvenience in software distribution. One possible solution is: upon the request for a copy of the software, a binary instance would be generated and shipped on-demand. Another way would be to maintain a binary repository for large-scale on-line distribution.

7 Related Work

7.1 Security through Diversity

Address Space Randomization (ASR) ASR is a technique which dynamically and randomly relocates a program's stack, heap, shared libraries, and even program objects. This is either implemented by an OS kernel patch [38], or modifying the dynamic loader [39], or binary code transformations [8], or even source code transformations [10]. The goal is to obscure the location of code and data objects that are resident in memory and foil the attacker's assumptions about the memory layout of the vulnerable program. This makes the determination of critical address values difficult if not impossible. Most ASR approaches cannot achieve data structure layout randomization, as the relative addresses of member variables do not get changed. Also, they need system support such as a loader kernel support, but we cannot assume that the remote system always has ASR. Even though the source code transformation approach [10] can to some extent generate polymorphic layout for static data in different runs, it still involves loader support, and does not randomize the variable member layout for dynamic data.

Instruction Set Randomization (ISR) ISR is an approach to preventing code injection attacks by randomizing the underlying system instructions [6, 28]. In this approach, instructions become data, and they are encrypted with a set of random keys and stored in memory. During program execution, a software translation is involved for decrypting the instructions before being fetched. ISR does not randomize any data structure layout.

Data Randomization Similar to ISR, program data can also be encrypted and decrypted. PointGuard [17] is such a technique which encrypts all pointers while they reside in memory and decrypts them only before they are loaded into CPU registers. It is implemented as an extension to the `gcc` compiler, which injects the necessary encryption and decryption wrappers at compilation time. Recently, Cadar et al. [12] and Bhatkar et al. [9] independently presented a new data randomization technique which provides probabilistic protection against memory exploits by XORing data with random masks. This is also implemented either as a C compiler extension or a source code transformation.

Operating System Interfaces Randomization Chew and Song proposed using operating system interface randomization to mitigate buffer overflows [13]. They randomized the system call mapping, global library entry points, and stack placement to increase the heterogeneity. Similarly, by combining ASR and ISR,

RandSys [27] randomizes the system service interface when loading a program, and at run-time de-randomizes the instrumented interface for correct execution.
Multi-variant System N-variant systems [18] are an architectural framework which employs a set of automatically diversified variants to execute the same task. Any divergence among the outputs will raise an alarm and can hence detect the attack. DieHard [7] is a simplified multi-variant framework which uses heap object randomization to make the variants generate different outputs in case of an error or attack. DieFast [32] further leverages this idea to derive a runtime patch and automatically fix program bugs. Reverse stack execution [36], i.e, reverse the stack growth direction, can prevent stack smashing and format string attacks when executed in parallel with normal stack execution in a multi-variant environment.

Compared with the above randomization approaches, DSLR exploits another randomization dimension with different goals, application contexts, and implementation techniques.

7.2 Data Structure Layout Manipulations and Obfuscations in Compilers

Propolice [22] is a `gcc` extension for protecting applications from stack-smashing attacks. The protection is implemented by a variable reordering feature to avoid the stack corruption of pointers.

There are several other data structure reorder optimizations in the compiler to improve runtime performance by improving data locality and reuse. Pioneering the approach is the one proposed by Hagog et al. [26] which is a cache aware data layout reorganization optimization in `gcc`. They perform structure splitting and field reordering to transform struct and class definitions. Recently, struct-reorganization optimizations have undergone the conversion from GIMPLE to Tree-SSA [25]. To handle multi-threaded applications (because of the false sharing), Raman et al. [34] proposed structure layout transformations that optimize both for improved spatial locality and reduced false sharing.

Similar to code optimizations to improve program performance, there exist code obfuscation techniques which aim to reduce the understand-ability of a program by reverse engineering. As data structures are important components and key clues to understand code, one of the most important obfuscations is data structure obfuscation. Common obfuscation techniques [15] include obfuscating arrays (such as splitting, regrouping [42], flattening, folding [14], and reordering arrays), obfuscating classes (such as splitting a class, inserting a new class, reordering class members), and obfuscating variables (such as substituting code for static data, merging and splitting variables [14]). These techniques are particularly useful to thwart the intermediate code analysis of Java and .NET, which tend to be easily analyzable [31].

Compared with these two approaches, DSLR has different goals. The reordering optimization techniques mentioned above aim to improve the performance, and their reordered layout is fixed/deterministic for all the compiled binaries. For data structure obfuscation techniques, the data structure layout they gener-

ate is again fixed. When taking snapshots of the memory to infer the signature, these techniques do not increase the diversity of data structure layout. However, we do not aim to obfuscate the data structure for a single binary. Instead, we aim to generate polymorphic layouts among multiple binary copies.

8 Conclusion

We have presented a new software randomization technique – DSLR – that randomizes the data structure layout of a program with the goal of generating diverse binaries that are semantically equivalent. DSLR can be used to mitigate malware attacks that rely on knowledge about the victim programs' data structure definitions. In addition, the simple implementation of DSLR poses a new challenge to data structure-based program signature generation systems. We have implemented a prototype of DSLR in `gcc` and applied it to a number of programs. Our evaluation results demonstrate that DSLR is able to achieve binary code diversity. Furthermore, DSLR is able to foil a number of kernel rootkit attacks by randomizing the layout of a key kernel data structure. Meanwhile, DSLR is able to reduce the similarity between binaries generated from the same source program.

9 Acknowledgment

We would like to thank Anthony Cozzie for his kind help with his Laika system for the evaluation of DSLR. We would also like to thank the anonymous reviewers for their very helpful comments. This research was supported in part by the US National Science Foundation (NSF) under grants 0546173 and 0716444. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the NSF.

References

1. Gnu compiler collection (gcc) internals. *http://gcc.gnu.org/onlinedocs/gccint/*.
2. Offensive computing. *http://www.offensivecomputing.net/*.
3. Using the gnu compiler collection (gcc). *http://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc/*.
4. Vx heavens. *http://vx.netlux.org/*.
5. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools (Second Edition)*. Addison-Wesley, 2006.
6. Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS'03)*, pages 281–289, New York, NY, USA, 2003. ACM.
7. Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation(PLDI'06)*, pages 158–168, New York, NY, USA, 2006. ACM.

8. Eep Bhatkar, Daniel C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, 2003.
9. Sandeep Bhatkar and R. Sekar. Data space randomization. In *Proceedings of International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'08)*, Paris, France, July 2008.
10. Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2005. USENIX Association.
11. Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and and Communications Security (CCS'07)*, 2007.
12. Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Philippe Martin, and Miguel Castro. Data randomization. *Technical Report MSR-TR-2008-120, Microsoft Research*, 2008.
13. Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. *Technical Report CMU-CS-02-197, Carnegie Mellon University*, 2002.
14. Seongje Cho, Hyeyoung Chang, and Yookun Cho. Implementation of an obfuscation tool for c/c++ source code protection on the xscale architecture. In *Proceedings of the 6th IFIP WG 10.2 international workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS'08)*, pages 406–416, Berlin, Heidelberg, 2008. Springer-Verlag.
15. Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. 1997.
16. Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *Proceedings of 2009 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.
17. Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.
18. Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: a secretless framework for security through diversity. In *Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.
19. Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *Proceeding of 8th Symposium on Operating System Design and Implementation (OSDI'08)*, December, 2008.
20. Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, Alexandria, VA, October 2008.
21. Weidong Cui, Marcus Peinado, Helen J. Wang, and Michael Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Proceedings of 2007 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2007.

22. H. Etoh. GCC extension for protecting applications from stack-smashing attacks (ProPolice). *http://www.trl.ibm.com/projects/security/ssp/*, 2003.
23. S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, page 67, Washington, DC, USA, 1997. IEEE Computer Society.
24. Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symposium (NDSS'03)*, February 2003.
25. Olga Golovanevsky and Ayal Zaks. Struct-reorg: current status and future perspectives. In *Proceedings of the GCC Developers' Summit*, 2007.
26. Mostafa Hagog and Caroline Tice. Cache aware data layout reorganization optimization in gcc. In *Proceedings of the GCC Developers' Summit*, 2005.
27. Xuxian Jiang, Helen J. Wang, Dongyan Xu, and Yi-Min Wang. Randsys: Thwarting code injection attacks with system service interface randomization. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS'07)*, pages 209–218, Washington, DC, USA, 2007. IEEE Computer Society.
28. Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS'03)*, pages 272–280, New York, NY, USA, 2003. ACM.
29. Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008.
30. Zhiqiang Lin and Xiangyu Zhang. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'08)*, Atlanta, GA, USA, November 2008.
31. Douglas Low. Protecting java code via code obfuscation. *Crossroads*, 4(3):21–23, 1998.
32. Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, USA, 2007. ACM Press.
33. Colin Percival. Naive differences of executable code. *http:/www.daemonology.net/bsdiff/*, 2003.
34. Easwaran Raman, Robert Hundt, and Sandya Mannarswamy. Structure layout optimization for multithreaded programs. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*, pages 271–282, Washington, DC, USA, 2007. IEEE Computer Society.
35. Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of Recent Advances in Intrusion Detection (RAID'08)*, pages 1–20, September 2008.
36. Babak Salamat, Andreas Gal, Alexander Yermolovich, Karthik Manivannan, and Michael Franz. Reverse stack execution. *Technical Report No. 07-07, University of California, Irvine*, 2007.
37. Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Guarantee Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'07)*, October 2007.
38. PaX Team. Pax address space layout randomization (aslr). *http://pax.grsecurity.net/docs/aslr.txt*.

39. XiaoFeng Wang, Zhuowei Li, Jun Xu, Michael K. Reiter, Chongkyung Kil, and Jong Youl Choi. Packet vaccine: Black-box exploit detection and signature generation. In *Proceedings of the 13th ACM Conference on Computer and Communication Security (CCS'06)*, pages 37–46, New York, NY, USA, 2006. ACM Press.
40. Gilbert Wondracek, Paolo Milani, Christopher Kruegel, and Engin Kirda. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008.
41. Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 260–269. IEEE Computer Society, 2003.
42. Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI'04)*, 2004.