Polymorphing Software by Randomizing Data Structure Layout

Zhiqiang Lin, Ryan D. Riley, and Dongyan Xu

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 layouts program data structure definitions and diversify the software that is compiled from the same suite of program source code. It can thwart data structure-based program signature generation system, and can also mitigate attacks which rely on knowing data structures, such as kernel rootkit attacks. We have implemented our polymorphism technique on top of the open source compiler collection `gcc-4.2.4` and applied it to a number of programs. Experimental results show that our data structure randomization can achieve software code diversity (with a rough instruction difference of 10%), cause false positives for a state-of-the-art data structure signature generation system, and provides diverse kernel data structures to mitigate a variety of kernel rootkit attacks.

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 first and foremost access and process these data structures. The 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, 19, 28, 38]. The knowledge of the layout of program data structures is valuable for both attack and defense scenarios.

Cyber attack – A buffer overflow attack, for example, is more likely to succeed if the attacker knows the program buffer is adjacent to a function pointer or return address [20]. In kernel rootkits, especially rootkits that manipulate kernel data directly, the attacker has to know (or assume) the layout of specific fields in order to manipulate them. In network application penetration tests, if the attacker knows the structure of the protocol message, he can reduce the fuzz space and speed up the test [37]. In reverse engineering, it is generally believed that program data structures contain a wealth of information for program understanding.

Cyber defense – In forensics analysis, the data structure layout associated with a memory dump is critical for acquiring meaningful and accurate results. In VMM-level introspection, kernel data structure definitions directly control the external interpretation of kernel events [26]. In network protocol analysis, the data structure associated with a protocol payload can be used to construct the exploit signature and match pre-

defined network traffic. In malware analysis, it has been reported recently that data structure layout can be used to generate malware signatures [18].

Forrest et al. [21] has suggested that monoculture computing is one of core reasons that computers are vulnerable to massive, replication attacks. Randomization could be introduced as a means to increase the diversity of nonfunctional aspects of software. This has already been widely exploited in address space randomization (ASR) [8, 10, 36, 39], instruction set randomization (ISR) [6, 27], data randomization [12, 16], and operating system interfaces randomization [13, 25]. Given the prior success of the strategy of randomization, we propose randomizing the program data structure layout.

In this paper, we demonstrate that we can polymorph software by randomizing the program data structure layout. We propose an approach to instrument a compiler (as compiler knows the program semantics) so that it will generate different program data structure layouts each time the source code is compiled. We note that this is a key difference when compared to obfuscation techniques [15] that are used in software protection and aim to make binary reverse engineering harder to uncover the data structure definition for a single piece of software. In our technique, we make it harder to derive data structure signatures out of multiple copies of the same software. Particularly, we instrument the compiler to scan all the encapsulated data structure definitions such as `struct` and `class` and then reorder their member layouts and insert garbage fields if necessary for each randomizable data structure.

Our data structure randomization technique provides a number of benefits to both cyber defense and cyber attack analysis. First, our technique can mitigate attack that rely on knowing the layout of data structures, such as kernel rootkit attacks. Second, our technique can make generalized forensic analysis and VMM introspection more difficult as the data structure definition is now specific to a certain copy of the software (e.g., a copy of the OS kernel). Finally, our technique can generate diversity in software to evade data structure-based signature systems.

We have implemented our data structure randomization technique on top of 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. The experimental results in Section 5 show that our technique can not only achieve software code diversity (with a rough instruction difference of %10), but also cause false positives for a state-of-the-art data structure signature detection system. Furthermore, it generates diverse kernel data structures to mitigate a variety of kernel rootkit attacks. Finally, our technique imposes very low performance overhead on `gcc` and the original un-randomized program (less than 1%).

2 Technical Challenges

In this section, we examine the major challenges faced in order to accomplish data structure randomization. the first challenge is to determine if a data structure is randomizable and the second challenge is the design of randomization techniques.

2.1 Randomizability of Data Structures

Program 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 in Laika [18]. We believe that a function's local variable layout can also be leveraged to compose signatures and thus also consider randomizing them.

However, randomizing any arbitrary data structure layout will not work in all cases. Some examples are: (1) If a data structure is used in network communication, all communicating parties must use the same definition. (2) If a data structure definition is public (e.g., defined in shared library `stdio.h`), it cannot be randomized. (3) There is 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 for 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. And these fields cannot be randomized, otherwise 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 programs outside software 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 a compiler can automatically spot all the randomizable data structures. In practice, however, only the programmer can have confidence that they are using a randomizable data structure. Moreover, even though we could have many heuristics to automatically spot those randomizable data structures, we cannot claim both completeness and safety. We can either guarantee safety without randomizing any data structures (no data structure is randomizable), or guarantee completeness by randomizing all of them (every data structure is randomizable). Hence, the accurate randomizable data structure set is the intersection of the safety set and completeness set. In this work, we require that programmers use new language key words to specify randomizable data structures.

2.2 Randomization Techniques

After the randomizable data structures are identified, the next challenge is how to randomize them. The simplest randomization scheme would be to reorder the layout. Our primary goal is to generate the polymorphic software, hence the more variation, the better. Therefore, we would like to design a randomization algorithm which uses an independent reorder scheme for each identified randomizable data structure. Suppose a program has n randomizable data structures and each structure has m members, then the maximum number of combinations for randomization would be $(m!)^n$.

However, if we aim to overcome the data structure signature approach proposed by Laika [18], reordering alone is still not sufficient. For example, suppose a structure has only two members which are both `int` types. No matter how we reorder these two members, the final shape for this structure is still composed with `int` and `int`. As a

result, to polymorph data structures containing several members of the same type, we have to use other randomization scheme. To this end, we insert garbage fields into these structures.

3 GCC Architecture and Our Design

Now that we have examined the challenges in data structure layout randomization, in this section we will present our system design. As C/C++ is commonly used in system and user level programming, we have implemented our polymorphism technique using `gcc`, one of the most popular compilers in UNIX world.

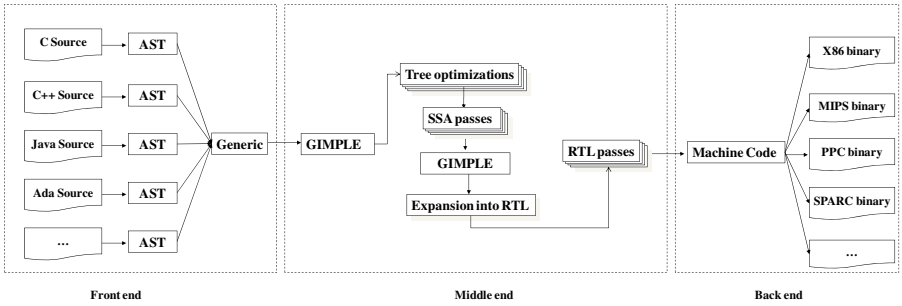

Fig. 1: An Overview of GCC Internal Architecture [1].

The GNU Compiler Collection (`gcc`) consists of over 2 million lines of code and has been in development for over 20 years. Through instrumenting `gcc` to reorganize the fields in encapsulated data structure definitions, our system will fill the memory image with a random layout each time the program source is compiled. Hence, we need to know where we should instrument `gcc`. Figure 1 illustrates an overview of the `gcc` internal architecture [1], which can be divided into three parts:

- **Front End** – it first reads and validates the syntax of each input program, then builds the initial Abstract Syntax Trees (AST). Because of the differences in languages such as C/C++, the format of the generated AST is slightly different among them, and there is one front end for each language. After building the AST, the last step in front end is to convert the language specific AST into a unified form called *generic*.
- **Middle End** – it takes care of most program analysis and optimizations. It first converts the *generic* AST 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) form of a 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 RTL optimization passes such as register allocation, code scheduling, and peepholes performed at the RTL level.

– **Back End** – finally, it generates the machine code for the target architecture based on the RTL representation. Examples of back-ends are X86, MIPS, SPARC, and so forth [1].

Given this internal architecture, the possible instrumentation points are AST, GIMPLE, SSA, and RTL. We decided to instrument the AST for the following reasons: (1) the AST retains much more original information from the program source code, such as type information and scope information for data structure and functions; (2) the AST representation is more understandable, 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 reconstruct the AST directly to reorder the member layout without any computation of the memory address.

Randomizable data structures 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 variable members in the data structure are chained together and represented by a link list. In order to achieve our goal, we can just capture the head node of the list, reorder the node of the list based on a random seed, and insert some random garbage into the list if necessary. This is the key aspect of our design and implementation.

Figure 2 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 declared sequential order, as shown in Figure 2(b). When compiled with our modified `gcc`, the order of the fields is randomized; we also add 2 garbage fields. Figure 2(c) shows the corresponding AST representation of `struct test`. Finally, when run the program compiled with our modified `gcc`, the layout of the instance of data structure `test` will be randomized according to the *random* order listed in the AST.

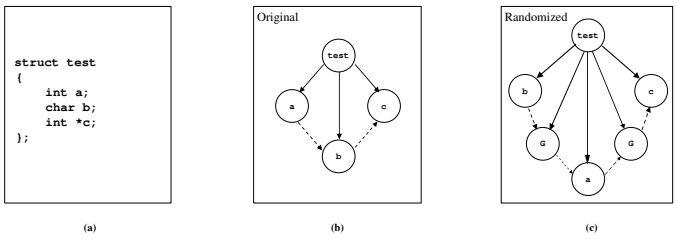

Fig. 2: Example of data structure randomization: (a) the original definition, (b) the original AST, and (c) the modified AST. The “G” node represents a garbage data structure we add to the structure. A dotted line represents the order of the member variables.

As discussed in Section 2, to enhance data structure layout diversity we adopt the following strategy: (1) different data structures at the same project building time will be reordered differently (with a different randomization seed), 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 the same data structure has a uniform layout across one entire build. As a result, under our strategy, suppose a program has two structures `S1`, `S2` which have 4 and 5 fields respectively. When we build the program using our modified `gcc`, structure `S1` and structure `S2` will be randomized differently. In addition, the same struct (e.g., `S1`) will have different layout in memory at distinctive project building time. Hence, the possible layout instance for this program would be $4! * 5!$ combinations. We believe such a strategy will greatly improve the polymorphism of the program, as the chance of generating the same instance will be $1/(\prod_{i=1}^{j} |S_i|!)$, where j represents the total number of randomizable data structures, and $|S_i|$ represents the total number of fields (members) for data structure S_i, for a given program.

4 Detailed Implementation

Our prototype is implemented on top of `gcc-4.2.4` with over one thousand lines of C code and we modified its AST representation to achieve our goal. It consists of four key components: (1) Keywords Recognizer, which recognizes the keywords we introduce to specify randomizability and garbage padding; (2) Reorderer, which reorders the field variables in a data structure definition according to a random seed; (3) Padder, which inserts the garbage fields to the data structure; and (4) Randomization Driver, which controls the polymorphism process according to the policy. In the remainder of this section, we present the details of their implementation.

```
...
<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. 3: A partial BNF illustration of our extend grammar for C/C++.

4.1 Keywords Recognizer

New Keywords Keywords are the words to identify particular syntactic forms in a programming language. We introduce several keywords to tell `gcc` how to randomize data structures.

The first keyword is named `__obfuscate__`. In fact, we implement it similarly to the way `__attribute__` is already implemented in [3]. Just like `__attribute__`, we offer options for `__obfuscate__` to tell `gcc` what kinds of obfuscation it should do, and we hence reserve two other keywords, `__reorder__` and `__garbage__`. The first one informs `gcc` that the object layout should be reorganized and the latter one informs `gcc` to insert some garbage fields into the object.

Grammar of the Keywords There are three types of randomizable data structures, which could be marked with `__obfuscate__` keyword, (1) C `struct`, (2) C++ `class`, and (3) stack variables declared in functions. The programmer is free to use the keywords as specified in the grammar shown in Figure 3. To better illustrate the usage of the keywords, we show some simple explicit examples in Figure 4.

```
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. 4: Example code: (a) shows how to randomize a class in C++, and (b) shows how to randomize a struct and stack variables inside the `main` function.

Implementation Details Since we perform our randomization at the AST level, there are two modifications when adding the new keywords. The first one is in lexical analysis, which makes the compiler recognize the new token, and the second one is to build our own *parser* for the keyword. As the C++ `class` implementation is quite similar to C `struct`, here we will only describe the C implementation.

- **Lexer** – There is an enumerate type called `rid` (*r*eserved *id*entifier) in the header file `c-common.h`. It contains all of the reserved keywords. Since we are adding our keywords as an extension of the GNU C extensions, we add the marks of our new keywords like "`RID_OBFUSCATE`, `RID_GARBAGE`, `RID_REORDER`" as new members of `rid`. Also we add a mapping between string and RID into "`struct reswords[]`" such as "{"`__obfuscate__`", `RID_OBFUSCATE`, `0` }" in `c-parser.c`.
- **Parser** – After we enable the compiler to understand the new keywords, we build a function called "`static tree c_parser_obfuscates (c_parser *parser)`" to take care of the randomization according to the policy variable `is_obfuscate`, which maintains a mapping from the keyword to the detailed policy and the corresponding randomization strategy. The mapping is shown in Table 1.

`is_obfuscate`	Keywords	Policy
0	ϵ	Do nothing
1	`__reorder__`	Reordering
2	`__garbage__`	Padding insertion
3	`__reorder__`, `__garbage__`	Both reordering and padding insertion

Table 1: Mapping of the keyword and the randomization policy.

4.2 Reorderer

When generating the program AST, `gcc` will chain the members of a particular data structure to a list. If we recognize the keywords `__reorder__`, we can then invoke our

Reorderer at the point where `gcc` finishes constructing the whole chain, and then we can reorder the members according to the random seed generated by our Randomization Driver.

We implement the Reorderer at different points for each category of randomizable data structures. To randomize a `struct`, we insert the Reorderer into the function `c_parser_struct_or_union_specifier`, which handles everything about `struct` or `union`, just after this function already constructs every item in `struct` or `union`. Note it is not necessary to randomize the members in a `union` as it only contains one instance of the declared members at run time. To randomize a `class`, we insert our Reorderer into function `unreverse_member_declarations`, and for local variables, we insert the Reorderer into function `c_parser_compound_statement_nostart`.

4.3 Padder

We implement the Padder to insert garbage between items of a data structure. The padder will be combined with the reorderer to finish the randomization, and it will be inserted in the same place as the Reorderer. When `gcc` recognizes the keyword `__garbage__`, the padder will insert variously sized garbage between items. This garbage creates noise in the memory image and makes it harder for a signature generation system to detect the data structure. The size of the garbage is determined by our Randomization Driver.

4.4 Randomization Driver

The Randomization Driver is the crucial component, as it controls the Reorderer and Padder and is directly related to the effectiveness of our polymorphism. 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 building file, if so, it directly takes that random value; otherwise it will generate the random value via the `glibc` function `random` and stores it in the project building file for future use. A project building file is a project wide specification file which contains the random value and the number of fields for each randomizable data struct, and is critical to ensure layout consistency across a single project build. In particular, when building projets such the Linux kernel and its drivers, it should use the same project building file, otherwise the kernel may use different data structure layouts then its drivers, which would likely cause crashes. Similarly, it checks whether the total number of elements of that data structure has been counted, otherwise it will count the total number of fields for that particular data structure and store it to the project building file as well.

After knowing the random value and total number of fields for a randomizable data structure, it takes two basic strategies to control the Reorderer and Padder.

Reordering Strategy Suppose our Randomization Driver gets a random value n, and counts the total number of elements for a particular data structure as m, and then it takes the reorder strategy shown in Algorithm 1 to perform the randomization.

In Algorithm 1, `pos[i]` represents the position for the member/field variable. For example, `pos[1]` represents the first member variable within the data structure, and

Algorithm 1 *Reordering Strategy*

```
1: Input: random value n, total number of member variables m, and the original order of the member variables: pos[1..m]
2: Output: the reordered member variables in pos'[1..m]
3: Initialization: j ← m;
4: Reorder(j, pos[1..j]){
5:     i ← n%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: }
```

`pos[m]` represents the last member variable. When we have the original order of the member variables, we recalculate the position of the member variable according to the random value n. For example, suppose $n = 26$, and $m = 4$ which has member variables {a,b,c,d}. There are 4 iterations in executing the algorithm.

1. i = 26%4 + 1 = 3; j = m = 4; move the 3^{rd} item c to pos'[4], and pos becomes {a,b,d}.
2. i = 26%3 + 1 = 3; j = 4 - 1 = 3; move the 3^{rd} item d to pos'[3], and pos becomes {a,b}.
3. i = 26%2 + 1 = 1; j = 3 - 1 = 2; move the 1^{st} item a to pos'[2], and pos becomes {b}.
4. i = 26%1 + 1 = 1; j = 2 - 1 = 1; move the 1^{st} item b to pos'[1], and return;

Thus, the final order of the member variables is {b,a,d,c}. We verify that Algorithm 1 can exactly achieve the goal of generating $m!$ polymorphic layout for a data structure containing m members.

Padding Strategy When we insert garbage between the member variables of an encapsulated data structure, the padding strategy determines the size of the garbage. We confine the size within the set of $\{8, 4, 2, 1\}$ bytes. We partition the random value n into four parts: x_1, x_2, x_3, and x_4, and each part has 8 bits. We then further reduce these 8 bits to 2 bits by calculating $x_i \bmod 4$ ($i \in \{1, 2, 3, 4\}$). These four random values fall into the range of 0 to 3. Then, we make 0 represent the size 8 bytes, 1 represent 4 bytes (4 byte garbage values will be made to look like pointers), 2 represent 2 bytes, and 3 represents 1 byte. Suppose there exists a structure which contains six member variables, and the four random value are 1, 3, 2, and 0. Then we insert 5 garbage fields between the items (how many garbage fields should be inserted could also be determined randomly; right now we just insert garbage in each field to maximum the difference), using the padding size of 4, 1, 2, 8, and 4 bytes, respectively. Note that if the data structure will be both reordered and padded, we first reorder and then insert garbage.

5 Experimental Results

In this section, we present our detailed evaluation results. We first assess the effectiveness of our randomization strategy in Section 5.1, and then measure the performance overhead of our approach on both `gcc` and the generated programs in Section 5.2.

5.1 Effectiveness

Data Structure Randomizability and Layout Diversity We applied our enhanced `gcc` to a number of goodware and malware programs. We use goodware such as openssh, and malware such as agobot which are collected from offensive computing [2] and VX Heavens [4]. We first measured the randomizability (i.e., how many data structures can be randomized) for each piece of software. Table 2 summarizes the results. We counted k_t ($t \in \{0, 1, 2\}$), which represents the total number of `struct`, `class`, and functions, respectively; and we counted the total number of corresponding randomizable data structure, j_t ($t \in \{0, 1, 2\}$), by manually examining each of `struct`, `class`, and functions. Hence, j_t/k_t represents the randomizability for `struct`, `class`, and functions which is shown in 3^{rd}, 4^{th}, and 5^{th} column in Table 2. Note that some of the function stack layouts which could not be randomized, the reason is that they contain `goto` statements (their label order is fixed). We also get the average value ($\overline{AVG_r}$) of randomizability for these data structures, which is shown in the 6^{th} column. We can see most of the data structures are randomizable (over 90%).

Benchmark program	LOC(K)	Randomizability of Data Structure				Possible Layout
		struct	class	funcs	$\overline{AVG_r}$	ω
42 Virus	0.88	1/1	-	24/24	100%	4E5
Slapper	2.44	26/30	-	69/70	92.6%	5E47
login-backdoor	2.81	3/3	-	44/44	100%	2E6
lyceum-2.46	3.97	10/13	-	30/30	88.5%	2E21
pingrootkit	4.81	26/27	-	57/57	98.2%	5E15
Mood-nt	5.31	36/37	-	121/122	98.2%	8E119
tnet-1.55	11.56	14/17	-	179/179	82.4%	7E82
Suckit	24.71	110/111	-	143/144	99.2%	9E159
samhain-1.28	45.08	57/61	-	438/438	96.7%	7E897
agobot3-pre4	245.44	23/31	50/50	340/346	86.2%	2E1106
patch-2.5.4	11.53	5/7	-	123/123	85.7%	4E3
bc-1.06	14.29	20/21	-	166/166	97.6%	6E56
tidy4aug00	15.95	9/18	-	341/341	75.0%	2E52
ctags-5.7	27.22	51/79	-	488/488	82.3%	3E668
openssh-4.3	76.05	63/80	-	820/838	88.3%	4E1271

Table 2: Evaluation Result of Randomizability and Layout Diversity.

Next, we measured the layout diversity for these programs. We first picked up one of the randomizable structures `bc_struct` in image instance of `bc-1.06`, to determine if we indeed randomized the layout. As shown in Figure 5, the data structure `bc_struct` compiled by our compiler (with a generated random number 669) has its layout changed significantly: not only has the order been randomized, it also contains an additional 6 garbage fields. We hence believe our system has achieved our design goal.

It is hard for us to enumerate all the possible layouts, but since we have counted the total number of randomized data structure j_t ($t \in \{0, 1, 2\}$) in each benchmark, then we can theoretically measure how many different variants our compiler can generate. The last column of Table 2 shows ω, which represents the total number of different instances our compiler could generate. Note $\omega = \prod_{i=1}^{j} |S_i|!$, where $j = \sum_{t=0}^{1} j_t$ represents the total number of randomizable `struct` and `class` structures (we exclude function as we already get very big numbers), and $|S_i|$ represents the total number of fields

(members) for data structure S_i. We can see the minimal number of variants for this software already exceeds 10^3.

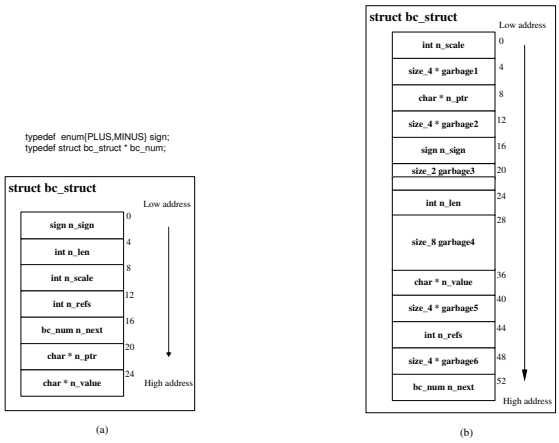

Fig. 5: Structure Layout Comparison: (a) Original Layout, and (b) Randomized Layout

Code Diversity A side effect of changing the data structure layout is that the assembly code generated will also be changed (i.e., we generate polymorphic code). The reason is the field variables in `structs` and `classes`, and even local variables, are accessed through data offsets which are changed by our randomization. Therefore, we would like to see how much code is different between our randomly generated code and the original un-randomized code.

To measure the code diversity, we first compile the benchmark program with unmodified `gcc` to get the original code, whose size is represented by I_0 shown in the 2^{nd} column of Table 3. Then we use our `gcc` to compile these programs, and for each one we generated three instances; their code sizes are represented by I_1, I_2, and I_3, respectively. Next, we compare the original binary code with the newly generated code using a tool called `bsdiff` [31] to get the difference represented by δ_i. It can be imagined that binary diff is a hard problem. We used `bsdiff`, a patch tool which generates the differences between the original program and modified program, in our evaluation. Compared with other binary diff tools, `bsdiff` adopts an algorithm named "approximate match", which counts the byte wise difference in two directions (both forward and backward) rather than one direction (often forward). As such the result generated by `bsdiff` is more precise and compact. Note that the results of `bsdiff` are highly compressible [31], and thus the difference result reported by `bsdiff` is relatively small. According to `bsdiff`, we can achieve almost 5-15% percent code difference. The last column of Table 3 $\overline{AVG_\delta}$ shows the average code difference for these three instances.

Evaluation Against Rootkit Process Hiding A kernel rootkit is a piece of malicious software that compromises a running operating system kernel. Usually an attacker will use them to hide his presence on a running system. An important feature of modern ker-

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%
login-backdoor	22.72	22.72	3.9	22.72	3.6	22.72	4.1	3.9%
lyceum-2.46	44.90	46.62	11.1	46.58	11.1	46.52	11.5	11.2%
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%
samhain-1.28	172.36	171.41	8.7	171.35	9.6	171.51	9.0	9.1%
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 3: Evaluation Result of Code Diversity.

nel 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 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 from rootkits. 6 different rootkits were tested to determine if they were able to hide a process under the randomized kernel. A summary of the results is available in Table 4. Detailed results for each kits 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 reliably 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 the `task_structs` 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. This is significant because there is very little work regarding preventing data-only rootkits, and data structure randomization is able to prevent the attack.

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 trouble 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 code.

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 the `task_struct`.

Most 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 sort of attack. Existing work [22,33,35] is able to effectively prevent these sorts of attacks. A different type of rootkit attack, however, is called a data-only attack. In this scenario a rootkit program will directly modify kernel data structures using a memory interface device such as `/dev/kmem`. The fuuld rootkit above employs this sort of attack. As evidenced by its effectiveness against the fuuld rootkit, however, data structure randomization is a very promising avenue of research to defend 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 significantly raises the bar for this type of attack.

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 4: Effectiveness of our approach against rootkit attack.

Evaluation against Laika The last effectiveness evaluation is against Laika, the data structure inference system. The released version of Laika only supports taking snapshot of Windows binaries, and we implemented our system on top of `gcc` which cannot compile Windows programs. To verify the effectiveness of our approach, we had to manually reorder the data structure for the tested windows program based on our randomization algorithm, and invoke the Windows compiler to finally generate the binary code. We picked 3 Windows programs, agobot, 7-zip, and notepad. For some reason,

Laika could not process the image file of notepad. Hence we only provide the result of 7-zip and agobot.

For each application, we generate 3 variant instances and use Laika to detect their layout similarity. In particular, Laika uses a mixture ratio [18] to classify the software, the closer the value is to 0.5, the more similar. Also, when detecting similarity, Laika has the option of filtering out pointers, we provide this result as well. Table 5 summarizes the result. Usually, the code difference for these images varied around 5%. For 7-zip, when filtering out pointers, Laika will use a mixture ratio of 0.502 on average to classify these image in the same family, and with pointers, it will use the mixture ratio of 0.511. It looks like these images do not appear significantly difference to Laika, and we examined the reason for 7-zip is that it only has 25 randomized structures, and more than 80 un-randomized structures which are in library. These library structures will dominate, and hence the mixture ratio is not too high. Agobot, however, contained 49 structures and 50 classes in its local code, so the mixture ratio becomes 0.57 without pointers, and 0.63 with pointers. From the mixture ratio, we can see we indeed have randomized the layout, and introduced noise to Laika. Also, even though for 7-zip we have high similarity, 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 structures) may use lots of similar library structures (such as the runtime support structure) in the implementation.

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 5: Evaluation result on Laika's effectiveness against our approach.

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
login-backdoor	1.1%	1.5%	1.7%	1.4%	N/A	N/A	N/A	N/A
lyceum-2.46	3.1%	3.8%	2.9%	3.3%	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
samhain-1.28	2.8%	1.2%	2.1%	2.0%	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 6: Normalized performance overhead.

5.2 Performance Overhead

The last evaluation is the performance overhead. Because we modified `gcc`, we would like to know how much overhead our approach imposed to `gcc`. We build the benchmark 3 times, and use the term g_1, g_2, and g_3 to represent the normalized gcc performance overhead of our approach. The 2^{nd}, 3^{rd}, and 4^{th} column of Table 6 show these results. We can see that on average our approach imposed around 2% performance overhead, which is mainly caused by structure random value lookup, field count, and the reordering.

Also, as we have changed the program data structure layout, which inevitably changes the program code, we would like to know the program performance overhead due to randomization. We measured the corresponding run-time overhead of the compiled binary. The 6^{th}, 7^{th} and 8^{th} column of Table 6 show these results, and we can see that randomization imposed roughly 1% overhead. We got the normalized overhead by running each program 10 times. Note for those virus and daemon malware programs, we did not measure its application performance overhead (appeared as N/A) because they are running in the background, and difficult to measure.

6 Limitations and Future Work

The first limitation is that right now we do not support other languages such as Java, because we instrument `gcc` at language specific AST level. Our next work includes either adding support to these languages, or studying the details of other `gcc` internal representation such as GIMPLE and RTL, and make our randomization more general.

The second limitation is that our randomizable data structure identification is not automatic and needs the programmer involvement. As we have discussed in Section 2, the fundamental challenge is the safety and completeness of spotting the randomizable data structures. To automate the identification, we could approximate the result by performing some sort of data flow analysis to identify those un-randomizable data structures such as those involved in network communications. For example, if we do not aim to achieve completeness (a reasonable assumption), we can adopt several heuristics to achieve our goal, such as using the execution context. If the structure is used in a network read or write, for example, then we know it cannot be randomized.

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 layout.

The last limitation is in the software distribution. When compiled by our `gcc`, a program can now has a large number of different copies. It will cause some inconvenience in the distribution. One way to handle this is upon requesting a copy of the software, one could generate a variant to distribute; another way would be to maintain such as a software pool for largely on-line distribution. To overcome the distribution issues, for us, we could think about a way to make the randomization mechanism self-contained, i.e., which takes effect at runtime and involves patching code generation in `gcc` to make

it generate the code which can dynamically change things like structure member offsets and still not violate the program semantics. In other words, we could generate identical copies of the binary, but it could contain a dynamic struct or class offset randomization mechanism. We leave this non-trivial code generation extension as another future task.

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 OS kernel patch [36], or modifying the dynamic loader [39], or binary code transformations [8], or even source code transformations [10]. Their 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 work cannot achieve our layout polymorphic goal, as the relative address of member variable does not get changed. Also, they need system software such as loader or kernel support, but the software vendor or attacker cannot assume the remote system has ASR support to foil the data structure signature system. To the best of our knowledge, 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 prevent "foreign" code injection attacks by randomizing the underlying system instructions [6, 27]. In this approach, instructions become data, and they are encrypted with a set of random keys and stored in memory. A software layer is responsible for decrypting the instructions before being fetched and executed by the processor. ISR does not randomize any data structure layout.

Data Randomization Similar to ISR, program data can also be encrypted and decrypted. PointGuard [16] 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 encrypt and decrypt 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 interfaces 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 [25] randomizes the system service interface when loading a program, and at run-time derandomizes the instrumented interface for correct execution.

Multi-variant System N-variant systems [17] is an architectural framework which employs a set of automatically diversified variants to execute the same task. Any diver-

gence 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 [30] further leverages this idea to derive a runtime patch and automatically fix program bugs. Reverse stack execution [34], 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 all of the above randomization approaches, we exploited another randomization dimension, and we have different goals, application context, and implementation techniques.

7.2 Data Structure Layout Manipulations and Obfuscations in Compilers

Propolice [20] 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. [24] 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 [23]. To handle multi-threaded applications (because of the false sharing), Raman et al. [32] proposed structure layout transformations that optimize both for improved spatial locality and reduced false sharing, simultaneously.

Similar to code optimizations to increase the performance of the program, there exist code obfuscation techniques which aim to reduce the readability of the 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 [40], flattening, folding [14], and reordering arrays), obfuscating classes (such as splitting a class, inserting 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, as they are easily analyzable [29].

Compared with our approach, we have different goals. For these reordering optimization techniques, they aim to improve the performance, and their reordered layout is fixed/deterministic for all the compiled program. For these data structure obfuscation techniques, again, the data structure layout they generate is fixed/deterministic. When taking snapshots of the memory to infer the signature, these techniques do not increase the polymorphism of data structure layout. However, we do not aim to obfuscate the data structure for single piece of software, and instead we aim to generate a polymorphic layout. Of course, we can benefit from these techniques to achieve more polymorphism such as using struct/class splitting.

8 Conclusion

We have presented a new type of software polymorphism by randomizing the program data structure layout with the goal of generating semantic equivalent but data structure layout different binary code. Our technique can thwart data structure based program signature generation systems used in malware defense, and can also mitigate attacks that rely on knowing data structure layouts such as data-only kernel rootkits. We have implemented our prototype on top of an open source compiler collection and applied it to a number of programs. Experimental results demonstrate our data structure randomization polymorphism can not only achieve software code diversity, but also foil the data structure signature generation system and provide a capability to mitigate a variety of rootkits that manipulate program (e.g., OS) data structures.

9 Acknowledgment

We thank Anthony Cozzie for his kind help with his Laika system in testing our Windows image samples.

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 *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. 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.
17. 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.
18. 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.
19. 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.
20. H. Etoh. GCC extension for protecting applications from stack-smashing attacks (ProPolice). *http://www.trl.ibm.com/projects/security/ssp/*, 2003.
21. 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.
22. 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.
23. Olga Golovanevsky and Ayal Zaks. Struct-reorg: current status and future perspectives. In *Proceedings of the GCC Developers' Summit*, 2007.
24. Mostafa Hagog and Caroline Tice. Cache aware data layout reorganization optimization in gcc. In *Proceedings of the GCC Developers' Summit*, 2005.
25. 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.
26. Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS'07)*, pages 128–138, New York, NY, USA, 2007. ACM.
27. 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.
28. 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.

29. Douglas Low. Protecting java code via code obfuscation. *Crossroads*, 4(3):21–23, 1998.
30. 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.
31. Colin Percival. Naive differences of executable code. *http:/www.daemonology.net/bsdiff/*, 2003.
32. 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.
33. 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.
34. Babak Salamat, Andreas Gal, Alexander Yermolovich, Karthik Manivannan, and Michael Franz. Reverse stack execution. *Technical Report No. 07-07, University of California, Irvine*, 2007.
35. 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.
36. PaX Team. Pax address space layout randomization (aslr). *http://pax.grsecurity.net/docs/aslr.txt*.
37. 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.
38. 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.
39. 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.
40. 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.