Misleading Malware Similarities Analysis by Automatic Data Structure Obfuscation

Zhi Xin, Huiyu Chen, Hao Han, Bing Mao, and Li Xie

State Key Laboratory for Novel Software Technology,
Department of Computer Science and Technology, Nanjing University
Nanjing 210093, China
{zxin.nju,mylobe.chen,hhinnju}@gmail.com, {maobing,lixie}@nju.edu.cn

Abstract. Program obfuscation techniques have been widely used by malware to dodge the scanning from anti-virus detectors. However, signature based on the data structures appearing in the runtime memory makes traditional code obfuscation useless. Laika [2] implements this signature using Bayesian unsupervised learning, which clusters similar vectors of bytes in memory into the same class. We present a novel malware obfuscation technique that automatically obfuscate the data structure layout so that memory similarities between malware programs are blurred and hardly recognized. We design and implement the automatic data structure obfuscation technique as a GNU GCC compiler extension that can automatically distinguish the obfuscability of the data structures and convert part of the unobfuscable data structures into obfuscable. After evaluated by fourteen real-world malware programs, we present that our tool maintains a high proportion of obfuscated data structures as 60.19% for type and 60.49% for variable.

Keywords: data structure, obfuscation, malware.

1 Introduction

To cope with the increasingly growing of malware, security vendors and analysts develop many different approaches to analyze various malicious code, such as virus, worm, trojan and botnet. Also, a number of research works have been focused on the arm race between obfuscation and deobfuscation techniques. Historically, the instruction-sequence-based signature [14] and static analysis methods [6,15,24] have been developed to recognize the characteristics of malicious code. This method indicates that a program is suspected to be infected by malware if a special sequence is found. However, Christodorescu et al. [8] present that these analysis methods can be disguised by various anti-reverse engineering transformation techniques, including code obfuscation [5,7], packing [21], polymorphism [17,22] and metamorphism [18,23]. In order to conquer the weakness of static analysis, security researchers adopt the dynamic analysis approaches [3,4,19] to log malware's relevant behaviors to enable the extraction of the core malicious functionality. However, malware can still hide their secret

M. Burmester et al. (Eds.): ISC 2010, LNCS 6531, pp. 181–195, 2011.

by employing trigger-based behaviors like bot-command, time-related and the presence of sandbox environment. Even though there are systems which conduct path exploration [9,10] against trigger-based behavior concealment, conditional code obfuscation [16] can still confuse the exploration by identifying and encrypting functionality branch variable.

Aiming to bypass previous malware's obfuscation techniques, Laika [2] explore the puzzle from a different aspect which focuses on the similarities derived from the runtime memory image, while the former approaches all concentrate on instructions sequences or behavior features, such as API sequence or parameters. First, Laika classifies every memory type to four basic categories: address, zero, string, and data (everything beside the former three classes). Then it adopts an Bayesian unsupervised algorithm to cluster the vectors of basic elements into objects and furthermore summarizes the categories out of the clustered objects.

According to the principle of Laika, we observe that the classification of byte vectors is based on the layout of basic elements. So if we can shuffle the field order of objects that of the same category in different memory image, the clustering algorithm will assort these objects to different classes. And the raising of obfuscation increased the obscurity of the similarities. Lin et al. [25] have exploited this weakness and present that data structure layout randomization can successfully disturb the procedure of similarity matching. However, their work have two obvious shortages: one is that the programmer have to mark the data structures manually by self-defined keywords, which will be a tedious task for large software program. And the other is that they summarized but give no solution to possible semantic errors caused by field reordering.

In this paper, we propose the *StructPoly*, a novel compiler extension solution based on GNU GCC, which is designed for overcoming the weaknesses of previous work. The biggest challenge in this work is to identify latent semantic errors caused by obfuscation and overcome unobfuscability as much as possible. And potential hazards can be classified mainly as four categories:(1) sequence dependent access; (2) link consistence; (3) system interface dependent; (4) special grammar. However, *StructPoly* can automatically obfuscate the data structure layout, distinguish the obfuscability of the data structures and convert part of the unobfuscable data structures into obfuscable. Details will be discussed in section 3. And also the "data structure" mentioned in this paper represent *struct* in C/C++. We evaluate our approach on fourteen real world malicious programs with the proportion of obfuscated data structures in average reaching 60.19% for type and 60.49% for variable. We outline our contribution as follows:

- We automatically identify the latent issues in which data structure obfuscation may cause semantic errors and conquer those puzzles by employing four diverse strategies. And the techniques could be applied to not only malware program but also all the other application programs.
- We implement the automatic data structure obfuscator as a extension of GCC compiler and convert part of the unobfuscable data structures into obfuscable.

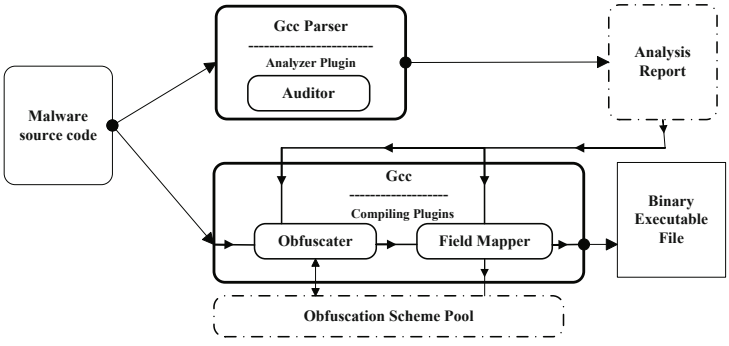

Fig. 1. The architecture of StructPoly system

Again, there is an arm race between malware author and security expert. One develops obfuscation techniques to tamper the precision of malware detectors, while the other advances new deobfuscation skill to infiltrate the "shell" of malware. From this angle, we improve the malware obfuscation methodology one step ahead to disguise the true face of malware against data structure similarity signature. Further more, other existing obfuscation techniques like polymorphic [17, 22] and metamorphic [18, 23] can be added on top of our obfuscation, making malware detection even harder.

The rest of the paper is organized as follows. We describe StructPoly's system overview in Section 2. Then Section 3 talks about identification of latent problems and Section 4 presents related technique implementation. And our experiments are evaluated and analyzed in Section 5. Section 6 reflects on our future work, followed by related works discussed in Section 7. Conclusion is given by Section 8.

2 System Overview

As Figure 1 presented, StructPoly processes the program in two steps. This is natural because essentially the compiler has to determine the order of data structure's fields before encountering any operations or expressions. This constraint forces our tool being separated into two parts: the parsing part and the obfuscating part.

In parsing part, the analyzer plugin *Auditor* parses the program and generate only the analysis report but not executable file. Substantially, the report will include all the *struct* categories which are against the criteria that describe the semantic errors under obfuscation. As an example in Figure 2, we present a latent problem class called *compound literal initialization* [32]. This syntax is based on the brace-enclosed initializer which is an easy form of initializing an object of aggregate type without allocating a temporary variable. To demonstrate the problem in Figure 2, we show a scenario with contrast between obfuscated and normal case. In the obfuscated case, the sequence *a*-*b*-*c* has been transformed

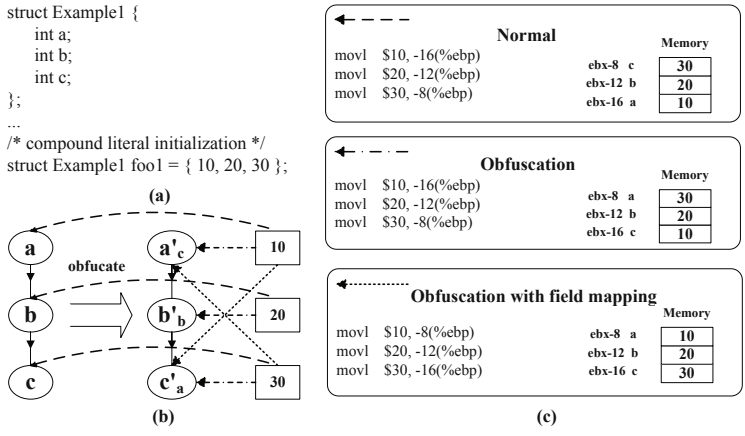

Fig. 2. The compound literal initialization. Part (a) presents the example of compound literal initialization. And part (b) describes the obfuscation and diverse assignment results in three context including "normal", "obfuscation" and "obfuscation with field mapping". Part (c) shows the binary operations and memory layout corresponded to all the three situations in part (b).

to *c*-*b*-*a* presented as $a_c^{'}$, $b_b^{'}$ and $c_a^{'}$ in part (b) and fields get error initialization value, such as *a* with *30* but not *10*. This is a mismatch between the obfuscated order of fields and the initialization value sequence following the data structure declaration. To address this issue, the *Auditor* should monitor all the compound literal initialization expressions. When data structure emerges, it records the type. With the concrete example, *Exmaple1* should be written into analysis report. The similar principle can be applies to other problems.

In the obfuscating part, we build two plugins into compiler. One is called *Obfuscater*, which do the obfuscation that means field reordering in this paper by referring to the analysis report generated previously. The other is *Field Mapper*, which is responsible for fixing the error in compound literal initialization by mapping values to correct fields whose position has been modified in the obfuscation. To the data structure classes recorded in the report, we can either obfuscate them or stay quiet. We leave the details of judgement strategies to next section and present the procedure in Figure 2 first. In the case, *Obfuscater* seeks the analysis report and realize that *Example1* can be obfuscated but need further assistance from Field Mapper. Then *Obfuscater* will reorder the fields of variable *foo1*, whose type is *Example1*, and transit it to *Field Mapper*. With learning *foo1*'s obfuscated layout in *Obfuscation Scheme Pool*, which store all the produced layouts of data structure types, Field Mapper adjusts the initialization values to match *foo1*'s new field sequence after obfuscation. The "obfuscation with field mapping" in Figure 2(c) shows the effect of this mapping: the fields all get right values even with obfuscation. In next section, we discuss the details of all the latent problems and present how to solve them.

3 The Set of Latent Problems

In this section, we discuss four categories problems and also present strategies to cope with them as summarized in Table 1.

Table 1. The set of latent problems and response strategies

Classification		Strategies
Sequence dependent access (A)	Compound literal initialization (a)	M
	Type conversion and pointer algorithm (b)	N
Link consistence (B)	Global variables and function arguments	E
API and library (C)	Posix standard interface, Glibc standard library, XSI extension and Linux special	N
Special grammar(D)	Flexible array field (GNU Gcc)	S

3.1 Sequence Dependent Access

Sequence dependent access means these accesses are all related to field sequence as declared. If program code access the fields with field names, there will be no mistakes (e.g. record.fieldname). But current sequence dependent accesses will fail after obfuscation in two kinds of situations: compound literals initialization and type conversion.

Compound literals initialization is a fixed order initialization way which can be applied to aggregate types like *array*, *struct* and *union*. From the discuss about this issue in Section 2, we can summarize that obfuscation indeed arouse assignment error in initialization however we can supply field mapping to regulate the assignment. We implement this regulation in two steps. First, we read the obfuscation scheme of data structure from "Obfuscation Scheme Pool" and then reversely adjust the order of initialization values. Finally, fields will be assigned with correct value even they have been disordered. The "Obfuscation with field mapping" in Figure 2 (c) shows the result. With the mapping, we convert part of the unobfuscable data structures into obfuscable. In Table 1, we mark this status as "M" which means "field mapping" will be the strategy for this situation.

Type conversion and pointer algorithm also triggers error. The *struct* pointer can be converted and assigned to a pointer related to naive type pointer (e.g. int*). Then the expression with offset calculation can be applied to index field in the structure. With no surprise, these expressions should be coded according to the field layout assumption as declared. As the structure variable *foo2* in Figure 3, the pointer *smallp* get its address and hope to index *a* field. But from the output with obfuscation in Figure 3, we can confirm that *smallp* just modify the field in unexpected location with expression **smallp = 1*. The target field should be the first field *a* but the last field *c* has been modified to *1* by mistake. Since nearly all the pointer-to analysis approaches cannot locate the source accurately to one certain field [34], it is unreasonable by using field mapping to adjust the

```
struct Example2 {
  int a;
  char b;
  int c;
};
...
struct Example2 foo2;
 foo2.a = 40;
 foo2.b = 'y';
 foo2.c = 60;
...
/* type conversion and pointer
arithmetic */
  int*   smallp = &foo2;

 *smallp = 1;
  char* second = smallp + 1;
 *second = "z";
```

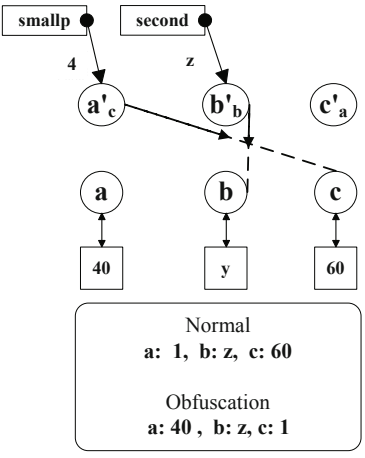

Fig. 3. Type conversion and pointer algorithm example

misplaced pointer algorithm. We realize that obfuscation is not suitable for this state of affairs and mark this strategy in Table 1 with "N", which means "No obfuscation" here.

3.2 Link Consistence

The link consistence problem occurs when the variables with the same type have diverse layout in different file. The issue emerges as two forms: external global variable and global function argument and the latter can be passed as value or pointer. As showed in Figure 4 (a), we obfuscated the structures in *Bar1.c* and meantime compile the *Bar2.c* normally. The global variable *foo1* has the normal field sequence, which can be observed from the ascending order of field address. But the value of fields are all mismatched, such as *b* with *1* but not *2*. Since different files are compiled within different compiling unit, with obfuscation, the operations related to a certain data structure will be fixed in different offsets and this could also be used to global function parameters. The symbol name resolution executed in linking will not affect this. So we constrain the same obfuscation scheme to variables with the same structure type distributed in different files. And we mark "E" in Table 1, which means "Enforce".

3.3 Operating System API and Standard Library Related

Except a few self-contained programs, programs always utilize the routines supplied by standard library or Application programming interface (API). Particularly, there are data structure parameters passed by pointer or value to library or API routines. As present in Figure 4 (b), the *struct tm* is used to represent time in standard library head file "time.h". The example code suppose to print the date. But with obfuscation, the output "30700/110/5" is obviously invalid as

Bar1.c

```
1  struct Inter {
2    int a;
3    int b;
4    int c;
5  };
6
7   struct Inter foo1;
8   ...
9   foo1.a = 1;
10 foo1.b = 2;
11 foo1.c = 3;
12 ...
```

Bar2.c

```
1  extern Inter foo1;
2  ...
3  printf("foo1: %d,  %d,  %d \n",
4    foo1.a, foo1.b, foo1.c);
5  printf("addr: %x,  %x,  %x \n"
6  , &foo1.a, &foo1.b, &foo1.c);

$:  Foo1 :  3,  1,  2
$:  addr :   8049704, 8049708, 804970c
```

(a)

```
1 #include <time.h>
2 #include <sys/time.h>
3
4 time_t mytm  =  time(NULL);
5 struct tm* ptm = localtime(&mytm);
6
7 printf("%d/%d/%d\n",  ptm->tm_year + 1900
8 , ptm->tm_mon, ptm->tm_mday);

$: 2010/5/24     ( Normal )
$: 30700/110/5  ( Output with obfuscation )
```

(b)

```
1 struct Example1 {
2   int a;
3   char b;
4   char c[];  // flexible array member
5 } bar = {10, 'a', "abcd"};

$: error: flexible array member not at end of struct
```

(c)

Fig. 4. The examples for link consistence, API related and flexible array member. The (a) part presents the problem exists in object file link. The (b) part shows the problem with data structure related to standard library. And the (c) part demonstrates that the last element shouldn't be moved in flexible array member case.

a date. This is because before a program is compiled, the library and operating system binary code already existed so that the operations related to those data structures parameters have been fixed as declaration in source code. Different from static link in "link consistence" issue, the nature issue related to this is that program has to accept the compiled code as dynamic linking or trap as a system call. So we must not obfuscate them otherwise there will be similar trouble like the case in Figure 4 (b).

3.4 Flexible Array Member

As a kind of special grammar, GNU GCC allows the last field of *struct* to be an uncertain size array called "flexible array member".(e.g. Figure 4 (c)). The field reordering may change the position of this field to cause error as *"error: flexible array member not at end of struct"*. We identify this situation and treat it specially to keep the last element motionless in the obfuscation procedure. "S" in the Table 1 means "Stillness".

4 Design and Implementation

4.1 Overview

GCC contains several intermediary representations (IR) including GENERIC, GIMPLE, SSA and RTL. GENERIC is an uniform representation for all

programming languages from different front ends. GIMPLE can be simplified from GENERIC, which is used for optimization and presents code as a 3-address form. We mainly instrument the process from C code to GENERIC for parsing part and field mapping because of the plenty of type information. Also, we modify an existing data-flow analysis technique called *type escape analysis* in GIMPLE form to identify type conversion and pointer algorithm.

4.2 Compound Literal Initialization

GCC supports versatile compound literals [20]. In the real world code, there are complex multiple nesting cases(e.g. task_struct in Linux kernel). We demonstrate the method of handling multiple nesting in Figure 5. The initialization algorithm organize the whole initialization as stack management called *constructor_stack*. Every time the nesting case goes deeper inside, function *push_init_level* will be called to push a new stack into *constructor_stack*, just like *Inter1* and *Inter2* inside the *Outer* in (a) in Figure 5. To deal with the stack management, we have two choices in implementation. The first one is to follow the GCC stack management process, which is like building a shadow stack; and the other is to unfold the initialization compound literal, which transforms nesting compound literal into a value sequence. We choose to implement the second scheme because it matches the GCC internal structure *tree* naturally. And after the compound literal has been unfolded, the value sequence linked in a chain can be assigned one by one with *TREE_CHAIN*. As the example presented in (c) of Figure 5, *StructPoly* regulates the 1-(2-3)-(4-5-6) sequence into (5-6-4-1-3-2) so it can be assigned to *foo*'s fields one by one.

There are several special situations which should be emphasized. The first thing is the *no-name structure*(e.g. struct {int a; int b;} bar;). When the structure type has no name, we can only identify it with *uid* given by GCC runtime symbol table. And the second puzzle is when there is no value for it in the

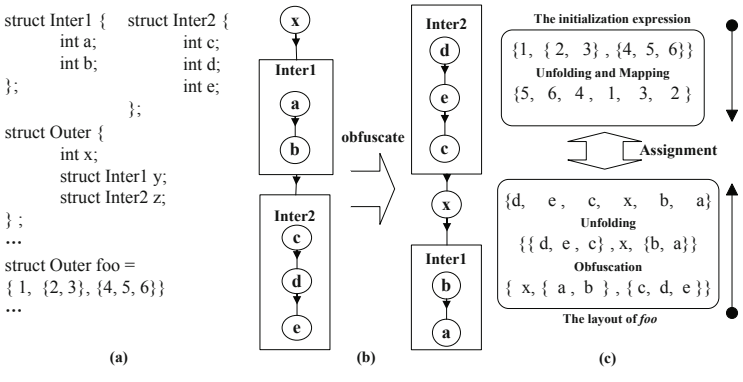

Fig. 5. The unfolding and mapping with complex compound literal initialization. Part (a) shows a nested compound literal initialization. Part (b) presents the field reordering scheme in Part (a) example. Part (c) shows the regulated value sequence.

compound constant expression, GCC compound literal initialization algorithm fill structure field with default value(e.g. struct {int a; int b; int c;} bar = {1,2}; c is assigned with default value). In this case, the mapping should be adjusted for the position of default value. And also there is another kind of initialization called "designated initializers" [20](e.g. struct foo = {.x = 1, .y = 2};). With structure field names, the elements can occur in any order. And we don't need to supply mapping in this case because field name will help element match the right field. But we do care about the mix case which has part of elements with field name and rest not. Right now, *StructPoly* cannot handle this mix situation and we will leave it as a future work.

4.3 Identify Type Conversion and Pointer Algorithm

StructPoly utilizes GCC's flow-sensitive interprocedural *type escape analysis pass* [20], which is used to identify the type conversion and pointer algorithm. The escaping types include pointer in a type conversion where the other type is an opaque or general type(e.g. char*). Usually, the type conversion happens between different pointer types. *StructPoly* intercepts type conversion in three expressions including explicit type conversion expression VIEW_CONVERT_EXPR (e.g. (int*) structp) and two implicit conversions as assignment expression MODIFY_EXPR (e.g. int* p = structp) and function argument types TYPE_ARG_TYPES. In all three cases, *StructPoly* identifies structure's pointer or multiple nested pointer to structure's pointer by find the essential data structure type with deep type exploration TYPE_MAIN_VARIANT. If either the to-type or from-type is structure type, marked as RECORD_TYPE, *StructPoly* identifies one structure related type conversion and record it.

4.4 Flexible Array Member

First, we identify this situation by verifying whether the last position field fits in the followed rules: (1) The field belongs to ARRAY_TYPE; (2) The TYPE_SIZE has to be null. Then we keep the flexible array member untouched and obfuscate the others.

4.5 Operating System API and Dynamic Link Library Related

We summarize 129 data structures defined as function parameter or reture value in *POSIX.1* (Portable Operating System Interface [for Unix]) [33] as a example. And all these data structures are used as a blacklist and stay away from obfuscation.

5 Evaluation

In this section, We evaluate our system with fourteen malware programs, including five viruses, four trojans, three worms and two botnets. The test environment is Linux Ubuntu 6.06 with Intel(R) Core(M) 2 Duo CPU E8400 3.00GHz and

Table 2. The details of obfuscability and overhead both in parsing and obfuscating

Benchmark malware	The obfuscability details					Extra overhead							
						in parsing				in obfuscating			
	α	β	γ	δ	ϵ	C_1(K)	C_2(K)	C_3(K)	AVG$_c$	M_1(K)	M_2(K)	M_3(K)	AVG$_m$
csmall	0	1	0	5	0	0.467	0.335	0.352	0.385	0.471	0.459	0.444	0.458
kaiowas10	0	0	0	2	0	0.180	0.176	0.184	0.180	0.075	0.081	0.077	0.078
vlpi	0	0	2	6	0	0.7	0.648	0.652	0.667	0.613	0.648	0.547	0.603
rapeme	0	1	0	4	0	0.299	0.271	0.287	0.286	0.148	0.137	0.187	0.157
lacrimae	0	1	0	28	0	1.104	1.19	1.054	1.116	1.702	1.699	1.612	1.671
lyceum2.46	0	0	0	10	0	2.747	2.517	2.469	2.578	1.06	0.814	0.995	0.956
utrojan	0	0	3	41	0	0.177	0.195	0.178	0.183	0.49	0.454	0.51	0.485
Q-1.0	0	0	0	5	0	2.441	2.398	2.584	2.474	0.07	0.068	0.15	1.140
vesela	0	1	2	30	0	0.254	0.242	0.236	0.244	0.78	1.469	1.145	0.096
wc	0	0	0	5	0	0.666	0.681	0.62	0.656	0.129	0.1	0.126	1.131
slapper	0	0	0	30	0	1.19	1.118	1.095	1.134	0.265	0.246	0.328	0.118
dover	2	0	3	31	0	1.331	1.145	1.236	1.237	0.262	0.226	0.27	0.253
BotNET1.0	0	1	0	24	0	2.177	1.867	1.923	1.989	0.083	0.085	0.087	0.085
toolbot	0	5	0	27	0	2.666	2.686	2.618	2.657	2.038	2.137	2.132	2.102

500M RAM. We evaluate our StructPoly system in three aspects: (1) we estimate the effectiveness in section 5.1 with both the scale of obfuscability and details of diverse situations; (2) then we measure the performance patched GCC in section 5.2; (3) finally we discuss the binary diversity of obfuscated malware programs against Laika in 5.3.

5.1 Effectiveness

The results of analysis report are given by parsing section of StructPoly in Table 2. It presents the details of diverse latent errors in compiled malware program. According to our strategies in Table 1, the amount of unobfuscable data structure types equals to the sum of β and δ, which stand for "Point algorithm" and "Operating system or library interface". And α, γ and ϵ respectively correspond to "Compound literal", "Link consistence" and "Special grammar", all of which bypass the latent traps. We can see from the results that *StructPoly* enables all the malicious programs completing their obfuscation safely and specially convert part of unobfuscable structures in worm *dover*, virus *vlpi* and trojan *utrojan* into obfuscable. And the most prevalent situation is "Operating system or library interface", which results in lots of the unobfuscable data structures.

Table 3 shows the consequences of obfuscability as a summary. We define *rType* as the sum of data structure types we obfuscated and *uType* as the amount of unobfuscable types. Also$rVar$ and $uVar$ define similar meanings to data structure variables. And "ScaleType" and "ScaleVar" represent the percentage of obfuscated data structure type and variable in total. They can be calculated by the formula: $ScaleType = \frac{rType}{rType+uType}$ $ScaleVar = \frac{rVar}{rVar+uVar}$. Comparing with previous work [2], StructPoly can obfuscate more data structures for two reasons:

(1) field mapping and the treatment of flexible array member transform some unobfuscable into obfuscable; (2) the exclusion of "Point algorithm" and "Operating system or library interface" makes obfuscation safer. The average scale with obfuscated type is 60.19% and 60.49% with variable. And the topmost has reached 81.20% in trojan *utrojan* for both type and variable.

5.2 Performance

In this section, we measure the performance overhead for both parsing and obfuscation part in *StructPoly*. As presented in Table 2, we use C_1, C_2 and C_3 to represent the individual overhead in parsing and M_1, M_2 and M_3 in obfuscation. In the end, *StructPoly* get performance overhead from 0.180% to 2.657% in parsing and producing analysis report. And in the obfuscating part, the performance overhead range from 0.078% to 2.102% which mainly are brought by field reordering, field mapping and mandatory order consistency.

Table 3. Obfuscability of malware programs

Benchmark	LOC(K)	rType	uType	rVar	uVar	ScaleType (%)	ScaleVar(%)
csmall	0.272	16	6	16	6	72.73	72.73
kaiowas10	0.147	6	2	6	2	75.0	75.0
vlpi	0.68	7	6	8	6	53.80	57.14
rapeme	0.63	6	4	7	4	60	63.64
lacrimae	6.072	24	10	24	10	70.60	70.60
lyceum2.46	3.59	46	28	53	28	62	65.43
utrojan	0.07	13	3	13	3	81.20	81.20
Q1.0	2.024	28	28	28	28	50	50
vesela	0.327	21	25	21	25	45.65	45.65
wc	2.324	47	31	47	31	60.26	60.26
slapper	2.441	49	30	50	30	62.25	62.5
dover	3.068	24	29	25	29	45.28	46.30
BotNET1.0	3.449	42	31	42	31	57.53	45.65
toolbot	6.191	31	30	31	30	50.82	50.82
AVG	2.235	26.77	18.79	26.5	18.79	60.19	60.49

5.3 Data Structure Obfuscation against Laika

Laika [2] represents the standard tool for data structure discovery from memory, which can only analyze the Windows image. But our *StructPoly* is based on GCC and cannot handle Windows program. So we cannot evaluate the effectiveness against Laika directly. However, certain previous work [25] has done this for us. They manually obfuscated the data structures in both benign programs and malware programs like 7zip and agobot. Laika measured similarity with a mixture radio. The similarity increase with the mixture radio closing to 0.5. The evaluation experience in *agobot* has presented the potency of data structure obfuscation. They obfuscated 49 data structures and made the mixture radio rise

to 0.63, which means Laika already has suspected the similarity between two variants. In our evaluation, we summarize the that virus is with average 17.4 data structures, which is less than 48 with trojan, 67 with botnet and 70 with worm. So we deduce that the *StructPoly*'s effect will be more notable in trojan, botnet and worm. And the obfuscated proportion of all the data structures has reached 60.19% and we speculate that the effectiveness will be obvious in high amount obfuscable data structures like worm *slapper* with 49, worm *wc* with 47 and trojan *lyceum-2.46* with 46.

6 Limitations and Future Work

Currently, we only complete obfuscation like field reordering but more specific approaches like garbage field inserting and struct splitting are under consideration. The inserting brings additional field into structure which is not declared to source code. So both the type and size are random with this technique. And splitting [35] method unbundles the fields that belong to one data structure type into multiple parts and connect them with data structure pointer. Obfuscating with a more fine granularity, they will significantly increase the diversity of binary code. The inserting expands the sample space of obfuscation, while the splitting separates the memory into more parts, which further tampers the tool like Laika that clusters trunk of bytes based on pointer.

7 Related Work

The program obfuscation and randomization techniques have been widely employed in many different aspects [26] including intellectual property protection, the defence of malicious code injection and the hiding of malicious code. From the intellectual property protection aspect, the encryption algorithm and keys as the vital parts of intellectual property are protected by code obfuscation [27] from reverse engineering. Additionally, they can also break program priori knowledge such as address layout [28, 29], which can thwart the malicious code injection attack. Finally, there are a number of existing techniques used to hide malicious code, including code obfuscation [5, 7], packing [21], polymorphism [17, 22] and metamorphism [18, 23]. Compared to previous works, our *StructPoly* shifts attention to runtime memory image rather than instruction sequence or program behavior.

There are a number of existing tools for data structure discovery or learning. Boomerang [30] as a decompiler takes input as an executable file and produces high level language representation of the program. But it can only recognize word-size data individually rather than a field of a certain data structure. Additionally, there are a number of memory-access-analysis methods [12,13,31] which convert the data structure layout from the patterns in the memory access. But all these recognition tool of memory access pattern cannot handle the similarity between different images. Also Lin et al. [1] present the binary type taint system, which infer the memory type by inferring from some known naive types from

special instructions, syscall arguments and library function arguments, but it also suffers from the absence of recognition to aggregate data structure type.

Malware detection techniques can be mainly classified as static and dynamic analysis. Historically, the instruction sequence based signature [14] and a number of static analysis methods [15, 6, 24] have been developed to recognize the characters of malicious code. Certainly, there are disadvantages with static analysis [8] so anti-malware vendors adopt the dynamic analysis approaches [3, 4] and unpackers [11]to penetrate the true face of malware under those "shells". After all these attentions concerned the similarities in malware's instructions or functionalities, Laika [2] compares the similarities between known malware and suspicious sample with runtime data layout in memory. However, our StructPoly automatically disturbs the data structure in the variants of malware, which can defeat the similarity identification mechanism.

8 Conclusion

We have implemented automatic data structure obfuscation as a novel polymorphism technique for malware programs so that the data structure signature of anti-virus system like Laika become more difficult to identify the similarity between obfuscated malware programs. We present that *StructPoly* as a extension of GCC compiler can automatically distinguish the obfuscability of the data structures and convert part of the unobfuscable data structures into obfuscable. With evaluated with fourteen real-world malware programs, we have experimentally shown that our tool can automatically obfuscate those malware programs and maintain a high scale of obfuscated data structures, which implies powerful potency against memory similar signature.

Acknowledgment(s)

This work was supported in part by grants from the Chinese National Natural Science Foundation (60773171, 61073027, 90818022, and 60721002), the Chinese National 863 High-Tech Program (2007AA01Z448), and the Chinese 973 Major State Basic Program(2009CB320705).

References

1. Lin, Z., Zhang, X., Xu, D.: Automatic Reverse Engineering of Data Structures from Binary Execution. In: Proceedings of the 17th Annual Network and Distributed System Security Symposium (2010)
2. Cozzie, A., Stratton, F., Xue, H., King, S.T.: Digging for Data Structures. In: The 8th USENIX Symposium on Operating Systems Design and Implementation (2008)
3. Anubis: Analyzing Unknown Binaries (2009), `http://anubis.seclab.tuwien.ac.at`
4. CWSandbox (2009), `http://www.cwsandbox.org/`

5. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM Conference on Computer and Communications Security (2003)
6. Christodorescu, M., Jha, S., Seshia, S.A., Songand, D., Bryant, R.E.: Semantics-Aware Malware Detection. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy (2005)
7. Popov, I.V., Debray, S.K., Andrews, G.R.: Binary obfuscation using signals. In: Proceedings of the 16th USENIX Security Symposium (2007)
8. Moser, A., Kruegel, C., Kirda, E.: Limits of Static Analysis for Malware Detection. In: 23rd Annual Computer Security Applications Conference (2007)
9. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D.X.: Automatically Identifying Trigger-based Behavior in Malware. In: Lee, W., et al. (eds.) Book chapter in Botnet Analysis and Defense (2007)
10. Moser, A., Kruegel, C., Kirda, E.: Exploring Multiple Execution Paths for Malware Analysis. In: Proceedings of the 28th IEEE Symposium on Security and Privacy (2007)
11. Coogan, K., Debray, S.K., Kaochar, T., Townsend, G.M.: Automatic Static Unpacking of Malware Binaries. In: The 16th Working Conference on Reverse Engineering (2009)
12. Balakrishnan, G., Reps, T.: Analyzing Memory Accesses in x86 Executables. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 5–23. Springer, Heidelberg (2004)
13. Balakrishnan, G., Reps, T.W.: DIVINE: Discovering Variables IN Executables. In: Proceeding of Verification Model Checking and Abstract Interpretation (2007)
14. Szor, P.: The Art of Computer Virus Research and Defense. Addison Wesley, Reading (2005)
15. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: Proceedings of the 16th USENIX Security Symposium (2003)
16. Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W.: Impeding Malware Analysis Using Conditional Code Obfuscation. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium (2008)
17. Pearce, S.: Viral polymorphism. VX Heavens (2003)
18. The Mental Drille Metamorphism in practice or How I made MetaPHOR and what I've learnt. VX Heavens (February 2002)
19. Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R.A.: Behavior-based spyware detection. In: Proceedings of the 15th Conference on USENIX Security Symposium (2006)
20. Stallman, R.: Using GCC: the GNU compiler collection reference manual. GNU Press (2009)
21. TESO. Burneye ELF encryption program (January 2004), `http://teso.scene.at`
22. Detristan, T., Ulenspiegel, T., Malcom, Y., von Underduk, M.S.: Polymorphic Shellcode Engine Using Spectrum Analysis. Phrack 61 (2003)
23. Julus, L.: Metamorphism. VX heaven (March 2000), `http://vx.netlux.org/lib/vlj00.html`
24. Kruegel, C., Robertson, W., Vigna, G.: Detecting Kernel-Level Rootkits Through Binary Analysis. In: Proceedings of the 20th Annual Computer Security Applications Conference (2004)
25. Lin, Z., Riley, R.D., Xu, D.: Polymorphing Software by Randomizing Data Structure Layout. In: Proceedings of the 6th SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (2009)

26. Balakrishnan, A., Schulze, C.: Code Obfuscation Literature Survey (2005), http://pages.cs.wisc.edu/~arinib/projects.htm
27. Colberg, Thomborson: Watermarking, Tamper-Proofing, and Obfuscation–Tools for Software Protection. IEEE Transactions on Software Engineering 28(8) (2002)
28. Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: Proceedings of the 14th Conference on USENIX Security Symposium (2005)
29. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: an efficient approach to combat a board range of memory error exploits. In: Proceedings of the 12th Conference on USENIX Security Symposium (2003)
30. Cifuentes, C., Gough, K.J.: Decompilation of Binary Programs. Software Practice & Experience (July 1995)
31. Ramalingam, G., Field, J., Tip, F.: Aggregate structure identification and its application to program analysis. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1999)
32. Status of C99 features in GCC, GNU (1999), http://gcc.gnu.org/c99status.html
33. Richard Stevens, W.: Advanced Programming in the UNIX Environment. Addison-Wesley, Reading (1992)
34. Shapiro, M., Horwitz, S.: The Effects of the Precision of Pointer Analysis. Lecture Notes in Computer Science (1997)
35. Collberg, C., Thomborson, C., Low, D.: A Taxonomy of Obfuscating Transformations. Technical Report 148, University of Auckland (1997)