Measuring the Robustness of Source Program Obfuscation - Studying the Impact of Compiler Optimizations on the Obfuscation of C Programs

Sandrine Blazy, Stéphanie Riaud

Measuring the Robustness of Source Program Obfuscation

Studying the Impact of Compiler Optimizations on the Obfuscation of C Programs

Sandrine Blazy
IRISA
University of Rennes 1
Sandrine.Blazy@irisa.fr

Stephanie Riaud
DGA-MI
INRIA
Stephanie.Riaud@inria.fr

ABSTRACT

Obfuscation is a commonly used technique to protect software from the reverse engineering process. Advanced obfuscations usually rely on semantic properties of programs and thus may be performed on source programs. This raises the question of how to be sure that the binary code (that is effectively running) is still obfuscated.

This paper presents a data obfuscation of C programs and a methodology to evaluate how the obfuscation resists to the GCC compiler. Information generated by the compiler (including effects of relevant optimizations that could deobfuscate programs) and a study of the disassembled binary code, as well as a dynamic analysis of the performances of binary code show that our obfuscation is worthwhile.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General—*Protection mechanisms*

Keywords

obfuscation, compiler optimization, anti-reverse engineering

1. INTRODUCTION

A well-known way of attacking a software is to start by reverse engineering it in order to reconstruct the memory layout using static or dynamic analysis tools operating over disassembled or binary code. Software reverse engineering is an active field of research and new reverse engineering techniques are regularly published. Obfuscating a program consists in transforming it into an equivalent program that is more difficult to understand or to analyze. Among the applications of obfuscation is the protection of software from reverse engineering. The numerous obfuscation techniques range from elementary syntactic techniques such as inserting opaque predicates to more sophisticated semantics-based techniques such as complicating control flows [3].

CODASPY'14, March 3–5, 2014, San Antonio, Texas, USA.
ACM 978-1-4503-2278-2/14/03.
http://dx.doi.org/10.1145/2557547.2557577.

Improvements in reverse engineering call for improvements in obfuscation. Obfuscation can be performed at different levels of program representation: at the source level, on any intermediate language or at the assembly or binary level. On the one hand, many obfuscations are performed directly at the binary or assembly level. It ensures that the code that is running is obfuscated. On the other hand, obfuscating a source program facilitates the design of advanced obfuscations relying on semantic properties of source programs.

Obfuscating a source program raises the question of how to be sure that the binary code (that is effectively running) is still obfuscated. Obfuscations can be classified according to two main categories, depending whether they mainly transform the control flow of a program or its data [3]. Many obfuscations related to control flow exist in the literature. Data obfuscations mainly hide data in memory and can also complicate the reverse engineering process.

This paper presents a data obfuscation of C programs and a methodology to evaluate how the obfuscation resists to the GCC compiler (including optimizations). This obfuscation is an example of obfuscation that is easier to design at the C level than at the assembly level. The obfuscation is hiding data structure fields at the source level, so that it will become much more difficult to reverse engineer the memory layout from a corresponding binary code. A detailed study of the information generated by the compiler (including effects of relevant optimizations that could deobfuscate programs) and of the disassembled binary code, as well as a dynamic analysis of the performances of obfuscated code show that our obfuscation is worthwhile.

2. BACKGROUND

We now present our data obfuscation and explain how the GCC compiler could possibly deobfuscate C programs.

2.1 A Data Obfuscation

Data obfuscations are usually classified according to the transformation they perform [3]. A data obfuscation aims at modifying the encoding of data, or the way data are stored, or the aggregation of data or the ordering of data. Our data obfuscation performs these four transformations.

It is detailed in Fig. 1 and replaces each field access having a scalar type (*e.g.* `t.a` in the example program of Fig. 1) by an address storing the current field value[1]. This address is computed by a function called `access` (that is added to the source program and detailed in Fig. 1). For instance,

[1]By default, all fields of C structures are replaced, but it is also possible to obfuscate only some selected fields.

the field `t.a` is replaced by the address `*access()`. Thus, the address of the current field is hidden in an array (called `access_array` in Fig. 1) that is accessed from the auxiliary function `access`. This access array groups together the whole field accesses of the program (*e.g.* its size is 4 in Fig. 1). Moreover, a new indirection is added. The access array is used to calculate indices of another array (called `address_array` in Fig. 1) storing the different field values.

Furthermore, the function `access` is exchanging the field values and updates accordingly the access array: at each function call (*i.e.* each field access in the original program) two field values are randomly chosen to be permutated.

```
struct test {int a;int b;};
int main (void){  /* Initial example program */
   struct test t;         int c;
   t.a =10;   t.b =20;    c = t.a;   return c + t.b;
}    /*-------------------------------------------------------*/
int access_counter;                          /* Obfuscated code */
int access_array[4] = { 0, 1, 0, 1, };
int *address_array[2];
int rand_a_b(int a, int b) { ... }  /* Returns an int in [a;b] */
int *access(){
    int nb_elem = 2, id1, id2, t, *ptr;
    id2 = rand_a_b(0, nb_elem);
    id1 = rand_a_b(0, nb_elem);
    t = **(address_array + id1);
    **(address_array + id1) = **(address_array + id2);
    **(address_array + id2) = t;
    ptr = *(address_array + id1);
    *(address_array + id1) = *(address_array + id2);
    *(address_array + id2) = ptr;
    access_counter = access_counter + 1;
    return *(address_array + *(access_array + (access_counter-1)));
}
int main(void)                    /* Obfuscated main function */
  struct test t;
  int c;
  access_counter = 0;
  *(address_array + 1) = &t.b;  *(address_array + 0) = &t.a;
  *access()= 10;                *access()= 20;
  c = *access();                return (c + *access());     }
```

Figure 1: An example of obfuscated program

Finding all field accesses in a given C program is more complex to perform at the assembly level, where the data flow is more difficult to interpret. It requires to consider all possible accesses via the registers, the stack and the rest of the memory. Thus, we designed our obfuscation at the C level that is better adapted than the assembly level.

2.2 Compiler Optimizations

Our obfuscation operates over C programs and we want to be sure that the corresponding binary code is as obfuscated as the source program. Consequently, we study the impact of the compilation on our obfuscation. We focus on the GCC compiler, as it is a widely used open compiler.

Among the great deal of compiler optimizations, we only study the relevant ones performing inverse transformations w.r.t. our obfuscation. As our obfuscation transforms scalar field accesses, we chose to study two kinds of optimizations: scalar reduction of aggregates (SRA) and scalar optimizations. SRA is the most relevant optimization as it transforms all the fields of a structure into different independent variables. Scalar optimizations rely on three basic optimizations: constant propagation, copy propagation and dead code elimination. We want to be sure that they neither transform the fields we obfuscate nor the statements that our obfuscation adds in the original program.

As other optimizations do, SRA selects candidate fields and rejects other fields of the program (*e.g.* a scalar field belonging to a structure having a pointer field). Then, among the candidate fields, SRA selects fields that will be changed and disqualifies the others (*e.g.* a field that is never accessed during program execution). Thus, only some of the field accesses are optimized by SRA. We then need to compare these fields with those of the original program. This impact analysis of SRA is also performed for scalar optimizations.

3. EVALUATION PROCESS

Our goal is to check that a binary obfuscated code is more difficult to reverse engineer than its corresponding initial program. Our method is general enough to be applied on other obfuscations operating over source programs. It follows the three main steps that are detailed in this section.

3.1 Impact Analysis of the Compilation

We compile separately the original program and the obfuscated program. Each compilation is performed twice: without any compiler optimization, as well as at the highest level of optimization. Then, we study the impact of each optimization on our obfuscation. Each optimization consists in different compiler passes (at least an intraprocedural and an interprocedural pass for the optimizations we have studied). For each pass, we analyze the optimization details generated by the compiler, in order to distinguish between the fields remaining obfuscated after the pass from the other optimized fields. For scalar optimizations, we also track the optimized expressions. Then, we compare these results with those of the original program. Our global measure is the conservation rate of field accesses in the obfuscated program.

3.2 Analysis of Disassembled Binaries

The next analysis takes into account the assembling and linking steps occurring after compilation, and checks that all the transformations added by the obfuscation (*e.g.* the permutation of values in an array) are still performed in the binary code. Using the interactive dynamic analysis of IDA Pro, the analysis compares the memory layout of the original and the obfuscated programs. We chose the IDA Pro tool because it is one of the most used tools for reverse engineering binary files.

The BinDiff tool built on IDA Pro is also used to compare two binary files and generate a similarity rate between both files [5]. We use it to study separately the effects of the optimization and the obfuscation, as well as the effects of the optimization on the obfuscation. Thus, we perform three comparisons with BinDiff: between the original program and the optimized program, between the original program and the obfuscated program, and between the obfuscated program and the optimized and obfuscated program.

3.3 Dynamic Binary Analysis

Instead of analyzing statically the impact of the compilation, we could perform a dynamic binary analysis. A static analysis is done once for each program, but a dynamic analysis is easier to implement, provided it scales to real-size source programs and does not generate huge execution traces that become impossible to interpret.

We used the Pin tool [10] to instrument obfuscated binaries and check whether the compiler has compiled away the obfuscation. Our Pin tool aims at collecting memory access to the access array added by our obfuscation, as well as the calls to the `access` function. Moreover, we also observed that the obfuscated binaries cannot be easily reverse engineered by such a dynamic binary instrumentation tool.

In addition, we reused our Pin tool to check that our obfuscation resists to a different compiler. We chose the Intel ICC compiler that is performing interprocedural optimizations that could modify obfuscated programs more than GCC can. The only difference we observed is the inlining of functions added by our obfuscation. But, the data structures remain the same in the obfuscated code, thus showing that compiled programs are still obfuscated.

Last, we also measured the increase of the execution time when obfuscating programs. Even if the main goal of an obfuscation is to have it resist reverse engineering analyses, the loss in performance cannot be completely neglected [1]. When the execution time of the obfuscated binary is much slower, the obfuscation is considered as unsatisfactory. Then, we chose to obfuscate less fields and we obfuscated again the initial program. This process was repeated until the execution times of the original and obfuscated programs become close. When the execution trace is small enough to be interpreted, another solution to update the obfuscation is to use the Pin tool in order to understand the discrepancy in the execution times of both programs and select the fields that will be obfuscated.

4. EXPERIMENTAL EVALUATION

Our benchmark for the impact analysis of the compilation is the GCC testsuite devoted to optimizations. Our results show that when a field is optimized in the original program, it is never optimized in the obfuscated program.

Next, the analysis of disassembled binaries shows that all the structure fields still exist in the obfuscated binary codes (even if the program was optimized), as well as the auxiliary variables and functions added by the obfuscation.

Last, we used the latest stable version of the GNU coreutils utility suite as input to our performance analysis. The size of programs ranges between 500 and 4800 lines. The execution times of all obfuscated programs were similar to those of their original programs, except for one program. We then used Pin to help us improving our obfuscation.

5. RELATED WORK

There is an abundant literature on control flow obfuscations. Data obfuscations were also designed mainly either to hide specific data by encrypting them (*e.g.* see [4]) or to diversify software by generating different binaries from a same source program [2, 9, 8]. Data obfuscations operate over C programs or over lower levels. For instance, the obfuscation defined in [9] randomizes the ordering of fields in C structures and inserts unused fields between them. Our obfuscation shares similarities with the obfuscation described in [1] that obfuscates both data location and data usage in order to protect against malware detectors. However, the transformations performed by their obfuscation differs from ours as the obfuscation operates directly over binary code.

Since the general criteria (*e.g.* resilience) defined in [3] for evaluating obfuscations, there is still no standard way of evaluating the effectiveness of an obfuscation. Evaluations based on static analysis exist [11, 6]. They rely on more advanced static analysis techniques than those of IDA Pro that we used, but to the best of our knowledge it is not clear that they scale on large programs such as those we tested.

6. CONCLUSION

We have presented a data obfuscation that hides field accesses in an array that is modified at each field access and hidden in a function. Indirect accesses to this array generate many pointers in obfuscated programs. Our obfuscation operates over C programs. We have showed that it is robust against compilation and very useful for programs manipulating structure fields storing information that need to be protected against reverse engineering tools.

Our experiments showed that reverse engineering of our obfuscated programs is very hard because of the code that is added by our obfuscation. It requires to analyze step by step a great deal of array accesses in order to replace many function calls by semantically equivalent array accesses.

We intend to conduct more experiments to test our obfuscation against some automatic deobfuscators relying on compiler optimizations such as those we have studied [7]. We also intend to design a program analyzer that will be able to automate some of the tedious tasks required to reverse engineer obfuscated programs.

7. REFERENCES

[1] A.Moser, C.Kruegel, and E.Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conf.*, pages 421–430, 2007.

[2] E. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proc. of USENIX Security Symp.*, pages 105–120, 2003.

[3] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, New Zeland, 1997.

[4] E. Damiani, S. De, C. Vimercati, and al. Computing range queries on obfuscated data. In *Proc. of the Information Processing and Management of Uncertainty in Knowledge-Based Systems*, 2004.

[5] H. Flake. Structural comparison of executable objects. In *Proc. of the Conf. DIMVA*, pages 161–173, 2004.

[6] R. Giacobazzi and I. Mastroeni. Making abstract interpretation incomplete: Modeling the potency of obfuscation. In *Static Analysis Symposium*, volume 7460 of *LNCS*, pages 129–145. 2012.

[7] Y. Guillot and A. Gazet. Automatic binary deobfuscation. *Journal in Computer Virology*, 6(3):261–276, 2010.

[8] T. Hagog. Cache aware data layout reorganization optimization in GCC. In *Proc. of the GCC Developers' Summit*, 2005.

[9] Z. Lin, R. D. Riley, and D. Xu. Polymorphing software by randomizing data structure layout. In *Proc. of the Conf. DIMVA*, pages 107–126, 2009.

[10] C.-K. Luk, R. Cohn, R. Muth, and al. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, 2005.

[11] A. Majumdar, A. Monsifrot, and C. Thomborson. On evaluating obfuscatory strength of alias-based transforms using static analysis. In *Conf. on Advanced Computing and Comms.*, pages 605–610, 2006.