Measuring the Robustness of Source Program Obfuscation - Studying the Impact of Compiler Optimizations on the Obfuscation of C Programs
Sandrine Blazy, Stéphanie Riaud

To cite this version:

HAL Id: hal-00927427
https://inria.hal.science/hal-00927427
Submitted on 13 Jan 2014
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 relevant optimizations that could deobfuscate programs) and a study of the disassembled binary code, as well as a dynamic analysis of 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].

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. 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 relevant optimizations that could deobfuscate programs) and of the disassembled binary code, as well as a dynamic analysis of 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 \( \text{address\_array} \)) in Fig. 1) that is accessed from the auxiliary function \( \text{access} \). 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 \( \text{address\_array} \)) storing the different field values.

Furthermore, the function \( \text{access} \) 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;
  t.a = 10; t.b = 20;
  c = t.a; return c + t.b;
 }

 int access_counter;

 int *access() {
  int *address_array[2];
  int access_array[4] = { 0, 1, 0, 1, };
  int access_counter = 0;
  *address_array[0] = &t.a; *address_array[1] = &t.b;
  *access() = 10; *access() = 20;
  return *(address_array + *(access_array + (access_counter-1))); }

 int main(void) /
  /* Obfuscated main function */
  struct test t;
  int c;

  *address_array[2][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))); }
```

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...
advanced static analysis techniques than those of IDA Pro based on static analysis exist [11, 6]. They rely on more transforming 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.

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


