OBfuscating C++ Programs Via Control Flow Flattening

T. Lásló and Á. Kiss
(Szeged, Hungary)

Abstract. Protecting a software from unauthorized access is an ever demanding task. Thus, in this paper, we focus on the protection of source code by means of obfuscation and discuss the adaptation of a control flow transformation technique called control flow flattening to the C++ language. In addition to the problems of adaptation and the solutions proposed for them, a formal algorithm of the technique is given as well. A prototype implementation of the algorithm presents that the complexity of a program can show an increase as high as 5-fold due to the obfuscation.

1. Introduction

Protecting a software from unauthorized access is an ever demanding task. Unfortunately, it is impossible to guarantee complete safety, since with enough time given, there is no unbreakable code. Thus, the goal is usually to make the job of the attacker as difficult as possible.

Systems can be protected at several levels, e.g., hardware, operating system or source code. In this paper, we focus on the protection of source code by means of obfuscation. Several code obfuscation techniques exist. Their common feature is that they change programs to make their comprehension difficult, while keeping their original behaviour. The simplest technique is layout transformation [1], which scrambles identifiers in the code, removes comments and debug information. Another technique is data obfuscation [2], which changes data structures,
e.g., by changing variable visibilities or by reordering and restructuring arrays. The third group is composed of control flow transformation algorithms, where the goal is to hide the control flow of a program from analyzers. These algorithms change the predicates of control structures to an equivalent, but more complex code, insert irrelevant statements, or “flatten” the control flow [3, 4].

Although nowadays several large software systems are written in C++, both open source and commercial obfuscator tools are mostly targeted for Java [5, 6]. Only a few tools are specialized for the C++ language [7, 8], and they only use trivial layout transformations. Since the importance of protecting C++ programs is not negligible, we have set out the goal to develop non-trivial obfuscation techniques for C++.

In this paper, we discuss the adaptation of a control flow transformation technique called control flow flattening to the C++ language. Although the general idea has been defined informally in [3], no paper has been published on the adaptation of the technique to a given programming language. The main contributions of this paper are the following:

• we have identified the problems of adapting the technique to C++ and we give solutions to them,
• we give the complete formal algorithm of the technique, and
• using a prototype implementation, we show the effect of the algorithm on test programs.

The remaining part of the paper is structured as follows. In Section 2 we give a detailed description of the problems that occurred during the adaptation of the technique to C++ and we offer solutions to them. Moreover, we also give the complete formal algorithm of the proposed technique. Next, in Section 3, we present our experimental results. In Section 4 we present an overview of the related works, and finally, in Section 5 we summarize our results and conclude the paper.

2. Flattening the control flow of C++ programs

In the case of most real life programs, branches and their targets are easily identifiable due to high level programming language constructs and coding guidelines. In such cases, the complexity of determining the control flow of a function is linear with respect to the number of its basic blocks [9]. The idea behind control flow flattening is to transform the structure of the source code in such a way that the targets of branches cannot be easily determined by static analysis, thus hindering the comprehension of the program.
The basic method for flattening a function is the following. First, we break up the body of the function to basic blocks, and then we put all these blocks, which were originally at different nesting levels, next to each other. The now equal-leveled basic blocks are encapsulated in a selective structure (a `switch` statement in the C++ language) with each block in a separate case, and the selection is encapsulated in turn in a loop. Finally, the correct flow of control is ensured by a control variable representing the state of the program, which is set at the end of each basic block and is used in the predicates of the enclosing loop and selection. An example of this method is given in Figure 1. The control flow graphs of the original and the obfuscated code show the change in the structure of the program, i.e., all the original blocks are at the same level, thus concealing the loop structure of the original program.

2.1. Difficulties in C++

According to the above description, the task of flattening a function seems to be quite simple. However, if it comes to the application of the idea to a real programming language, then we come across some problems. Below we will discuss the difficulties we faced during the adaptation of control flow flattening to the C++ language.

As the example in Figure 1 already presented, breaking loops to basic blocks is not equal to simply splitting the head of the loop from its body. Retaining the same language construct, i.e., `while`, `do` or `for`, in the flattened code would lead to incorrect results, since a single loop head with its body detached definitely cannot reproduce the original behaviour. Thus, for loops, the head of these structures has to be replaced with an `if` statement where the predicate is retained from the original construct and the branches ensure the correct flow of control by assigning appropriate values to the control variable.

Another compound statement that is not trivial to handle is the `switch` construct. The cause of the problem in this case is the relaxed specification of the `switch` statement, which only requires that the controlled statement of the `switch` is a syntactically valid (compound) statement, within which case labels can appear prefixing any sub-statement. An interesting example which exploits this lazy specification is Duff’s device [10], where loop unrolling is implemented by interlacing the structures of a `switch` and a loop. A slightly modified version of the device and its possible flattened version are given in Figure 2.

When it comes to loops and `switch` statements, we cannot omit to discuss unstructured control transfers either. If left unchanged in the flattened code, `break` and `continue` statements could cause problems, since instead of terminating or restarting the loop or `switch` they were intended to do, they would restart the control loop of the flattened code. To avoid this, such instructions have to
Figure 1. The effect of control flow flattening on the source code (a: original, b: flattened) and on the control flow graph (c: original, d: flattened).
Obfuscating C++ programs via control flow flattening

```c++
int swVar = 1;
while (swVar != 0)
{
    switch (swVar)
    {
    case 0: do {
        switch (cnt % 4)
        {
        case 0: do
        {
            *to++ = *from++;
        case 3: *to++ = *from++;
        case 2: *to++ = *from++;
        case 1: *to++ = *from++;
        } while ((cnt -= 4) > 0);
        break;
        }
        case 1:
        {
            *to++ = *from++;
            *to++ = *from++;
            *to++ = *from++;
            *to++ = *from++;
            swVar = 3;
            break;
        }
        case 2:
        {
            L1: *to++ = *from++;
            L2: *to++ = *from++;
            L3: *to++ = *from++;
            L4: *to++ = *from++;
            swVar = 2;
            break;
        }
        case 3:
        {
            if ((cnt -= 4) > 0)
            swVar = 2;
            else
            swVar = 0;
            break;
        }
        break;
    }
}
swVar = 0;
break;
}
```

(a) (b)

Figure 2. Duff’s device (a: original code, b: flattened version)

be replaced in the flattened program by assignments to the control variable in a way that the correct order of execution is ensured. Figure 3 gives an example of this replacement.

Compared to C, C++ introduced an additional control structure, the try-catch construct for exception handling. By simply applying the basic idea of control flow flattening to a try block, i.e., determining the basic blocks and placing them in the cases of the controlling switch would violate the logic of exception handling. In such a case, the instructions that would be moved out of the body of the try would not be protected anymore by the exception handling mechanism, and thrown exceptions could not be caught by the originally intended handlers. To keep the original behaviour of the program in the flattened version, try blocks have to be flattened independently from the other parts of the program resulting in a new while-switch control structure, which remains under the control of the try construct. Thus, the flattening of try constructs produces multiple levels of flattened blocks. This causes problems again when an unstructured control transfer has to jump across different levels.

Figure 4 shows an example of the multiple levels of flattened blocks yielded
by the transformation of a try construct, as well as a solution for jumping across
levels when it is required by a break statement. Although using goto statements
is usually discouraged by coding guidelines, there are cases when their use is

2.2. The algorithm of control flow flattening

In the following, we will propose an algorithm for flattening the control flow
of C++ functions, which solves the problems presented in the previous sub-
section. The algorithm expects that the abstract syntax tree of the function-to-
be-flattened is available, and after a preprocessing phase, it traverses the tree in
one pass, along which the obfuscated version of the function is generated.

In the formal description of the algorithm, see Figures 5, 6, and 7, the bold
words mark the keywords of the used pseudo-language, the formalized parts are
typeset in roman font, while the parts which are easier to explain in free text are
in italic. The output of the algorithm is a C++ code, for which typewriter font
and double quotes are used. Throughout the algorithm, two symbols are used
additionally: ⊕ denotes string concatenation, while ⇒ outputs the result of the
algorithm, e.g., to the console or to a file.

The algorithm starts at the control_flow_flattening procedure, see Figure 5,
which first performs a preprocessing on the function. In this step, all the variable
declarations that are not at the beginning of the function, i.e., the ones that are
preceded by other statements, are eliminated to avoid visibility problems,
that would result from the change in the scope of such declarations. So, the
Obfuscating C++ programs via control flow flattening

```cpp
while (1) {
    try {
        int swVar1 = 1;
        L: while (swVar1 != 0) {
            switch (swVar1) {
                case 1:
                    if (1)
                        swVar1 = 2;
                    else
                        swVar1 = 0;
                    break;
                case 2:
                    try {
                        int swVar2 = 1;
                        while (swVar2 != 0) {
                            switch (swVar2) {
                                case 1:
                                    buf = new char[512];
                                    break;
                                case 3:
                                    cerr << "exception" << endl;
                                    swVar1 = 1;
                                    break;
                            }
                        }
                    } catch (...) {
                        swVar1 = 3;
                    } catch (...) {
                        swVar1 = 3;
                    } catch (...) {
                        swVar1 = 3;
                    } break;
            }
        }
        buf = new char[512];
        break;
    } catch (...) {
        swVar1 = 1;
    } catch (...) {
        swVar1 = 3;
    }
    break;
}
}
```

(a) (b)

Figure 4. Exception handling with unstructured control transfer (a: original code, b: flattened code).

declaration of these variables is moved to the beginning of the function, and only their initialization is left in place, i.e., converted to an assignment. Possible name collisions are resolved by variable renaming.

Although moving variable declarations to the beginning of the function is an important topic, its complexity [12] and the limits of the paper make it impossible to give a formal solution for this problem here. Thus, in the following, we assume that the preprocessing step has already been performed and the variable declarations are separated from the rest of the function body.

The actual flattening starts at the procedure `flatten_block`, where the construct controlling the control flow is generated. As Figure 4 presented in the previous subsection, sometimes it is necessary to jump across different levels of flattened blocks. To aid this, the controlling loop is annotated with a label, and this label
The algorithm of control flow flattening, part one.

Figure 5. The algorithm of control flow flattening, part one.
procedure transform_if (if_stmt, entry, exit)
begin
switch_variable := top(levels).variable
then_entry := unique_number()
else_entry := if_stmt has an else branch ?
    unique_number() : exit
⇒ case * ⋄ entry ⋄ * : (*
for each label in labels of if_stmt do
⇒ label ⋄ * ;:
endfor
⇒ * if (" ⋄ predicate of if_stmt ⋄ ")*
⇒ * " ⋄ switch_variable ⋄ " = " ⋄ then_entry ⋄ * ;:
⇒ * " else" ⇒ * " ⋄ switch_variable ⋄ " = " ⋄ else_entry ⋄ * ;:
⇒ * " break;"
⇒ *)
transform_block(true branch of if_stmt, then_entry, exit)
if if_stmt has an else branch then
transform_block(false branch of if_stmt, else_entry, exit)
endif
end

procedure transform_while (while_stmt, entry, exit)
begin
switch_variable := top(levels).variable
body_entry := unique_number()
⇒ case * ⋄ entry ⋄ * : (*
for each label in labels of while_stmt do
⇒ label ⋄ * ;:
endfor
⇒ * if (" ⋄ predicate of while_stmt ⋄ ")*
⇒ * " ⋄ switch_variable ⋄ " = " ⋄ body_entry ⋄ * ;:
⇒ * " else" ⇒ * " ⋄ switch_variable ⋄ " = " ⋄ exit ⋄ * ;:
⇒ * " break;"
⇒ *)
transform_block(true branch of while_stmt, body_entry, entry)
push(continues, (size(levels), exit))
transform_block(body of while_stmt, body_entry, entry)
pop(continues)
pop(continues)
end

procedure transform_switch (switch_stmt, entry, exit)
begin
switch_variable := top(levels).variable
⇒ case * ⋄ entry ⋄ * : (*
for each label in labels of switch_stmt do
⇒ label ⋄ * ;:
endfor
⇒ * switch (" ⋄ predicate of switch_stmt ⋄ ") {
⇒ for each case label in cases of switch_stmt do
⇒ goto_label := unique_identifier()
⇒ ⇒ " ⋄ goto_label ⋄ ;:
⇒ ⇒ goto " ⋄ goto_label ⋄ ;;
⇒ add a label named goto_label to the
⇒ statement referenced by case_label
endfor
⇒ *)
⇒ *)⇒ " ⋄ switch_variable ⋄ " = " ⋄ exit ⋄ * ;:
⇒ " break;"
⇒ *)
push(breaks, (size(levels), exit))
transform_block(body of switch_stmt, unique_number(), exit)
pop(breaks)
end

procedure transform_do (do_stmt, entry, exit)
begin
switch_variable := top(levels).variable
body_entry := unique_number()
⇒ case * ⋄ entry ⋄ * : (*
for each label in labels of do_stmt do
⇒ label ⋄ * ;:
endfor
⇒ * if (" ⋄ predicate of do_stmt ⋄ ")*
⇒ * " ⋄ switch_variable ⋄ " = " ⋄ body_entry ⋄ * ;:
⇒ * " else" ⇒ * " ⋄ switch_variable ⋄ " = " ⋄ exit ⋄ *
⇒ " break;"
⇒ *)
push(breaks, (size(levels), exit))
push(continues, (size(levels), entry))
transform_block(body of do_stmt, body_entry, entry)
pop(breaks)
pop(continues)
end

Figure 6. The algorithm, part two.
procedure transform_for(for_stmt, entry, exit)
begin
  switch_variable := top(levels).variable

  test_entry := unique_number()
  body_entry := unique_number()
  ⇒ "case " ⊕ entry ⊕ " : {" ⇒ "break;" ⇒ "}"
⇒ "case " ⊕ test_entry ⊕ " : {" ⇒ "if (" ⊕ predicate of for_stmt ⊕ ")" ⇒ " " ⊕ switch_variable ⊕ " = " ⊕ body_entry ⊕ ";" ⇒ " else" ⇒ " " ⊕ switch_variable ⊕ " = " ⊕ exit ⊕ ";" ⇒ "break;" ⇒ "}" ⇒ "case " ⊕ inc_entry ⊕ " : {" ⇒ " " ⊕ increment part of for_stmt ⇒ " " ⊕ switch_variable ⊕ " = " ⊕ test_entry ⊕ ";" ⇒ "break;" ⇒ "}" push(breaks, size(levels), exit)) push(continues, size(levels), inc_entry)) transform_block(body of for_stmt, body_entry, inc_entry) pop(breaks) pop(continues) end

procedure transform_try(try_stmt, entry, exit)
begin
  switch_variable := top(levels).variable
  ⇒ "case " ⊕ entry ⊕ ": {" ⇒ "for each label in labels of try_stmt do ⇒ label ⊕ "⇒ "break;" ⇒ "}"
⇒ "for each handler in catch handlers of try_stmt do ⇒ "catch (" ⊕ parameter of handler ⊕ ") {" flatten_block(body of handler) ⇒ "}"
⇒ " end
end

procedure transform_sequence(sequence, entry, exit)
begin
  ⇒ "case " ⊕ entry ⊕ ": {" ⇒ "for each stmt in sequence do ⇒ for each label in labels of stmt do ⇒ label ⊕ "⇒ "break;" ⇒ "}"
case type of stmt of continue:
  ⇒ "case " ⊕ entry ⊕ ": {" ⇒ "if top(continues).level < size(levels) then ⇒ "goto " ⊕ levels[top(continues).level].label ⊕ ";" ⇒ "else ⇒ "break;" endif
end
break:
  ⇒ "case " ⊕ entry ⊕ ": {" ⇒ "if top(breaks).level < size(levels) then ⇒ "goto " ⊕ levels[top(breaks).level].label ⊕ ";" ⇒ "else ⇒ "break;" endif
otherwise:
  ⇒ "stmt endcase endfor ⇒ top(levels).variable ⊕ " = " ⊕ exit ⊕ ";
⇒ "break;" ⇒ "}" end

Figure 7. The algorithm, part three.
The last type of compound statements to be transformed is `try`. As discussed in the previous subsection, this construct requires the use of multiple levels of flattened blocks. Thus, contrary to the previous procedures, `transform_try` in Figure 7 calls `flatten_block` recursively instead of `transform_block`.

Finally, the procedure `transform_sequence` is the one that handles simple statements, and this is where the stacks managed in `flatten_block` (levels) and in some of the `transform` procedures (`breaks`, `continues`) are utilized. All `break` and `continue` statements are rewritten to an assignment to the control variable, more precisely, to the appropriate control variable. The `levels` stack together with either the `breaks` or the `continues` stack determine which variable is to be used. Additionally, if the stacks indicate that the control has to cross levels of flattening, a `goto` instruction is inserted, as presented in the example in Figure 4.

3. **Experimental results**

We implemented a prototype version of the algorithm discussed in the previous section using the CAN C++ analyzer of the Columbus framework [13]. To evaluate the effects, we executed the prototype on a benchmark, which consisted of 23 functions selected from the Java-is-faster-than-C++ Benchmark [14], the C version of the LINPACK Benchmark [15] and LDA-C [16].

To measure the effect of control flow flattening on comprehendability, we computed McCabe’s cyclomatic complexity metric [17] for each function before and after applying the transformation to them. The results show a significant, 3.95-fold increase in complexity, on average, with a maximum multiplier of 5 and a minimum of 2, see Tab. 1. As Figure 8 displays, the effect of the algorithm scales linearly as the original complexity increases.

In addition to the effect on complexity, we measured the effect of control flow flattening on resource consumption as well. We examined two attributes of the functions: their size and their runtime. The size of the functions was measured by counting the number of nodes in the abstract syntax tree (AST), while the runtime data was computed by compiling the benchmark programs using GCC for x86 target and extracting information from profiles gathered on a Linux-based PC running at 3 GHz. The results, listed in Tab. 2, show that on average, both size and runtime doubled. However, if flattening is not applied to the whole program but only to some selected functions, as expected from real applications, the effect on total size and runtime can be much smaller.
Table 1. The effect of control flow flattening on complexity

<table>
<thead>
<tr>
<th>Function</th>
<th>Original Complexity</th>
<th>Flattened Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>main (sumcol.cpp)</td>
<td>3</td>
<td>15 (5.00 ×)</td>
</tr>
<tr>
<td>mmult (matrix.cpp)</td>
<td>4</td>
<td>20 (5.00 ×)</td>
</tr>
<tr>
<td>main (almabench.cpp)</td>
<td>4</td>
<td>20 (5.00 ×)</td>
</tr>
<tr>
<td>save_lda_model (lda-model.c)</td>
<td>3</td>
<td>15 (5.00 ×)</td>
</tr>
<tr>
<td>new_lda_model (lda-model.c)</td>
<td>3</td>
<td>15 (5.00 ×)</td>
</tr>
<tr>
<td>log_sum (utils.c)</td>
<td>2</td>
<td>9 (4.50 ×)</td>
</tr>
<tr>
<td>read_data (lda-data.c)</td>
<td>4</td>
<td>17 (4.25 ×)</td>
</tr>
<tr>
<td>matgen (linpack.cpp)</td>
<td>7</td>
<td>28 (4.00 ×)</td>
</tr>
<tr>
<td>deep (penta.cpp)</td>
<td>5</td>
<td>20 (4.00 ×)</td>
</tr>
<tr>
<td>gen_random (random.cpp)</td>
<td>1</td>
<td>4 (4.00 ×)</td>
</tr>
<tr>
<td>radecdist (almabench.cpp)</td>
<td>2</td>
<td>8 (4.00 ×)</td>
</tr>
<tr>
<td>digamma (utils.c)</td>
<td>1</td>
<td>4 (4.00 ×)</td>
</tr>
<tr>
<td>argmax (utils.c)</td>
<td>3</td>
<td>12 (4.00 ×)</td>
</tr>
<tr>
<td>dgefa (linpack.cpp)</td>
<td>16</td>
<td>62 (3.88 ×)</td>
</tr>
<tr>
<td>main (moments.cpp)</td>
<td>5</td>
<td>19 (3.80 ×)</td>
</tr>
<tr>
<td>lda_mle (lda-model.c)</td>
<td>5</td>
<td>19 (3.80 ×)</td>
</tr>
<tr>
<td>main (nestedloop.cpp)</td>
<td>9</td>
<td>34 (3.78 ×)</td>
</tr>
<tr>
<td>main (matrix.cpp)</td>
<td>3</td>
<td>11 (3.67 ×)</td>
</tr>
<tr>
<td>main (sieve.cpp)</td>
<td>8</td>
<td>27 (3.38 ×)</td>
</tr>
<tr>
<td>main (random.cpp)</td>
<td>3</td>
<td>9 (3.00 ×)</td>
</tr>
<tr>
<td>anpm (almabench.cpp)</td>
<td>3</td>
<td>9 (3.00 ×)</td>
</tr>
<tr>
<td>main (wc.cpp)</td>
<td>9</td>
<td>25 (2.78 ×)</td>
</tr>
<tr>
<td>ack (ackermann.cpp)</td>
<td>3</td>
<td>6 (2.00 ×)</td>
</tr>
</tbody>
</table>

Figure 8. Relationship between the complexities of the original and the flattened code.
Table 2. The effect of control flow flattening on program size and runtime.

<table>
<thead>
<tr>
<th>Function</th>
<th>Size (AST)</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>main (sumcol.cpp)</td>
<td>94 → 154 (1.64 ×)</td>
<td>1.93 → 1.98 (1.04 ×)</td>
</tr>
<tr>
<td>mmult (matrix.cpp)</td>
<td>61 → 162 (2.66 ×)</td>
<td>50.51 → 111.65 (2.21 ×)</td>
</tr>
<tr>
<td>main (almabench.cpp)</td>
<td>90 → 187 (2.08 ×)</td>
<td>0.12 → 0.56 (4.67 ×)</td>
</tr>
<tr>
<td>save_lda_model (lda-model.c)</td>
<td>103 → 181 (1.76 ×)</td>
<td>0.00 → 0.00 (1.00 ×)</td>
</tr>
<tr>
<td>new_lda_model (lda-model.c)</td>
<td>77 → 150 (1.95 ×)</td>
<td>0.01 → 0.01 (1.00 ×)</td>
</tr>
<tr>
<td>logsum (utils.c)</td>
<td>39 → 77 (1.97 ×)</td>
<td>6.19 → 9.39 (1.52 ×)</td>
</tr>
<tr>
<td>read_data (lda-data.c)</td>
<td>198 → 285 (1.44 ×)</td>
<td>0.01 → 0.02 (2.22 ×)</td>
</tr>
<tr>
<td>matgen (linpack.cpp)</td>
<td>126 → 263 (2.09 ×)</td>
<td>0.72 → 1.19 (1.65 ×)</td>
</tr>
<tr>
<td>deep (penta.cpp)</td>
<td>79 → 177 (2.24 ×)</td>
<td>16.58 → 33.33 (2.01 ×)</td>
</tr>
<tr>
<td>radecdist (almabench.cpp)</td>
<td>92 → 127 (1.38 ×)</td>
<td>1.10 → 1.28 (1.16 ×)</td>
</tr>
<tr>
<td>digamma (utils.c)</td>
<td>81 → 92 (1.14 ×)</td>
<td>53.64 → 52.32 (0.98 ×)</td>
</tr>
<tr>
<td>argmax (utils.c)</td>
<td>34 → 91 (2.68 ×)</td>
<td>0.05 → 0.29 (5.80 ×)</td>
</tr>
<tr>
<td>dgefa (linpack.cpp)</td>
<td>494 → 810 (1.64 ×)</td>
<td>0.64 → 0.67 (1.05 ×)</td>
</tr>
<tr>
<td>main (moments.cpp)</td>
<td>105 → 197 (1.88 ×)</td>
<td>0.59 → 0.59 (1.00 ×)</td>
</tr>
<tr>
<td>lda_zole (lda-model.c)</td>
<td>101 → 195 (1.93 ×)</td>
<td>0.02 → 0.03 (1.50 ×)</td>
</tr>
<tr>
<td>main (nestedloop.cpp)</td>
<td>89 → 268 (3.01 ×)</td>
<td>96.87 → 377.48 (3.90 ×)</td>
</tr>
<tr>
<td>main (matrix.cpp)</td>
<td>112 → 166 (1.48 ×)</td>
<td>0.01 → 0.01 (1.00 ×)</td>
</tr>
<tr>
<td>main (sieve.cpp)</td>
<td>93 → 228 (2.45 ×)</td>
<td>45.39 → 98.20 (2.16 ×)</td>
</tr>
<tr>
<td>main (random.cpp)</td>
<td>56 → 93 (1.66 ×)</td>
<td>2.70 → 8.29 (3.07 ×)</td>
</tr>
<tr>
<td>anpm (almabench.cpp)</td>
<td>27 → 60 (2.22 ×)</td>
<td>0.64 → 1.24 (1.94 ×)</td>
</tr>
<tr>
<td>main (wc.cpp)</td>
<td>99 → 224 (2.26 ×)</td>
<td>39.51 → 43.08 (1.09 ×)</td>
</tr>
<tr>
<td>ack (ackermann.cpp)</td>
<td>24 → 34 (1.42 ×)</td>
<td>77.52 → 111.43 (1.44 ×)</td>
</tr>
</tbody>
</table>

4. Related works

The scientific literature on program obfuscation is about ten years old. A significant paper is written by Collberg, Thomborson and Low [18], which describes the importance of obfuscation, and summarizes the most important techniques, mainly for the Java language. They give a classification of the described techniques and define a formal method to measure their quality. In a later work [19], they focus on the obfuscation of the control flow of Java systems by inserting irrelevant, but opaque predicates in the code. In their paper they prove that this method can give effective protection from automatic deobfuscators, while it does not increase code size and runtime significantly. In another paper [2], they describe a way of transforming data structures in Java programs. A summary of their results is given in [1] by Low, and a Java-targeted implementation is presented as well.

Similarly to Collberg et al., Sarmenta studies parameterized obfuscators in [20]. The parameters can select the parts of the program where transformation will be applied, or even the transformations that will be applied. Additionally, the transformations themselves can have parameters, too. Sarmenta investigates the combination of encryption and obfuscation as well. E.g., encrypted functions can be obfuscated or encryption can be performed during obfuscation.
In his PhD thesis, Wroblewski discusses low (assembly) level obfuscation techniques [21]. In his work, he analyzes and compares the main algorithms of the field, and based on the results, he gives the description of a new algorithm. Zhuang et al. developed a hardware-assisted technique [22], which obfuscates the control flow information dynamically by on-the-fly changing memory accesses thus concealing recurrent instruction sequences from being identified. Ge et al. present another dynamic approach [23] where control flow obfuscation is based on a two-process model: the control flow information is stripped out of the obfuscated program and a concurrent monitor process is created to contain this information. During the execution of the program process, it continuously queries the monitor process thus following the original path of control.

Wang et al. describe an obfuscation technique [3] which combines several algorithms, e.g., data flow transformation and control flow flattening. They show that the problem of analyzing and reverse engineering the code obfuscated using their technique is NP-complete. Unfortunately, neither do they give the description of the algorithm for control flow flattening nor discuss how to adapt it to a specific language. Chow et al. investigate control flow flattening in [4], too, but they claim that they approach works for programs containing simple variables and operators and labelled statements only.

Code obfuscation is not only discussed in scientific papers, but is utilized in several open source and commercial tools. Most of these tools are targeted for Java, and work on byte code, e.g., Zelix Klassmaster [5], yGuard [6] and Smokescreen [24]. These tools perform name obfuscation (renaming of classes, methods and fields), encode string constants, and transform loops using gotos. The renaming technique is used by the Thicket tool family [8] and COBF [7] as well. Thicket supports several programming languages, while COBF is the only C/C++ obfuscator freely available.

The later tool was the only one we could compare to our prototype implementation. Even though it transforms the names of classes, functions and variables, and removes spaces and comments from the source thus making the code unreadable for a human analyzer, this gives no protection against automatic de-obfuscators. We evaluated COBF on the benchmark functions but, as expected, we observed no change in the McCabe metric after obfuscation. What is more, in some cases the renamings that COBF applied caused compile time errors.

5. Summary and future work

We realized the need for the obfuscation of C++ programs, and thus we adapted a technique called control flow flattening. As the main contribution of
this paper, we identified the problems that occurred during the adaptation and proposed solutions for them. Moreover, we also gave the formal description of an algorithm that performed control flow flattening based on these solutions. The algorithm shows how to transform general control structures and how to deal with unstructured control transfers. Additionally, the technique flattens exception handling constructs as well. Since the transformed control structures are quite similar in other widespread languages as well, the algorithm can be used as a starting point when control flow flattening has to be adapted. Finally, we implemented a working prototype of the algorithm. The results of its evaluation were presented, which showed that the complexity of programs increased significantly due to the obfuscation.

During the development of the algorithm and its implementation we identified several possibilities for future work. First of all, we realized that moving variable declarations to the beginning of functions is important for the correctness of the technique. However, the limits of the current paper does not allow to elaborate on this topic in full detail. Thus, we discuss it only informally, and focus on the formalization of the transformation of the control flow. Still, in a future work, we would like to take a closer look at the problem.

In addition to the above, there are other ways, too, to enhance control flow flattening. A simple but effective approach is to permute the order of the flattened blocks, thus moving related blocks away from each other. Moreover, using goto and labels only instead of the while-switch construct we can further harden the comprehension of the obfuscated code. Another method is to obfuscate the values assigned to the control variable, in a way that they are not compile time constants anymore, or to use alias variables to make static analysis more difficult. In the future, we plan to extend our current implementation with these features since, as proven in [3], control flow flattening combined with aliasing can render the determining of the precise control flow NP-hard. Finally, we also plan to evaluate the runtime implications of the algorithm in a real case study and look for enhancements if needed.

References


T. László and Á. Kiss
University of Szeged
Department of Software Engineering
Árpád tér 2.
H-6720 Szeged, Hungary
laszlo.timea@stud.u-szeged.hu, akiss@inf.u-szeged.hu