Annales Univ. Sci. Budapest., Sect. Comp. 30 (2009) 3-19

OBFUSCATING C++ PROGRAMS VIA CONTROL FLOW FLATTENING

T. László 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 occured 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 contruct 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.

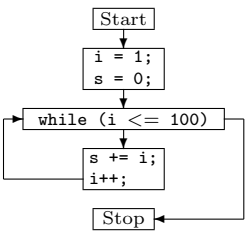
When it comes to loops and `switch` statements, we cannot omit to discuss unstructured control transfers either. If left unchanged in the flattened code, `break` and `continue` statements could cause problems, since instead of terminating or restarting the loop or `switch` they were intended to do, they would restart the control loop of the flattened code. To avoid this, such instructions have to

```
i = 1;
s = 0;

while (i <= 100) {

  s += i;
  i++;

}
```

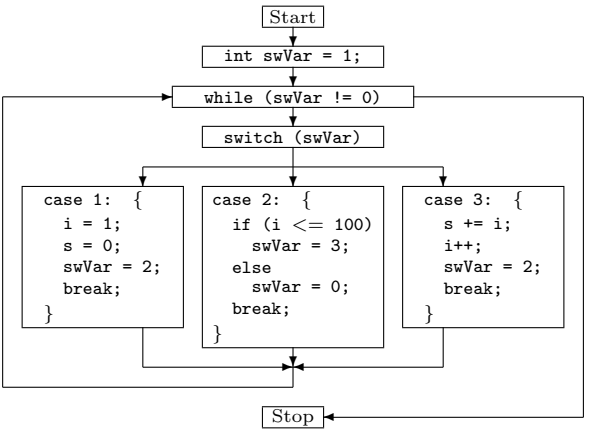
(a)

```
int swVar = 1;
while (swVar != 0) {
  switch (swVar) {
    case 1:  {
      i = 1;
      s = 0;
      swVar = 2;
      break;
    }
    case 2:  {
      if (i <= 100)
        swVar = 3;
      else
        swVar = 0;
      break;
    }
    case 3:  {
      s += i;
      i++;
      swVar = 2;
      break;
    }
  }
}
```

(b)

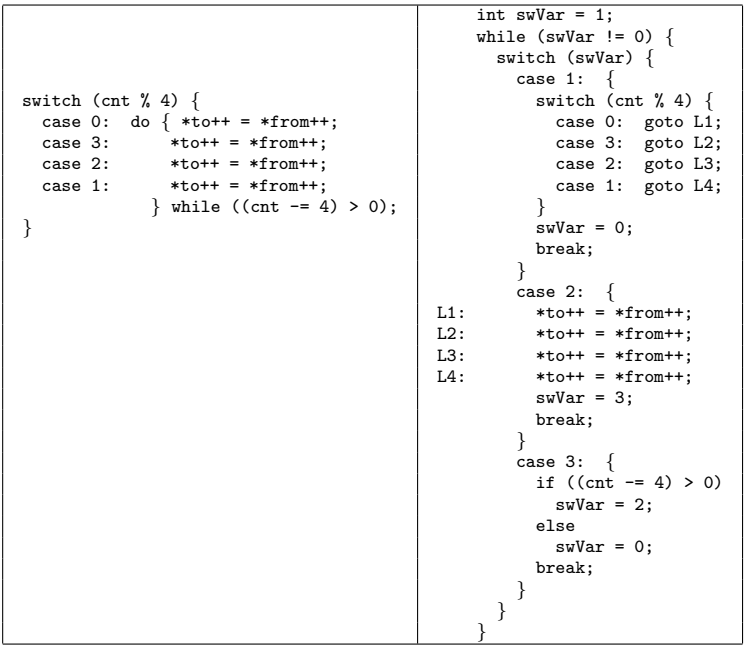


(c)



(d)

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).



```
switch (cnt % 4) {
  case 0:  do { *to++ = *from++;
  case 3:       *to++ = *from++;
  case 2:       *to++ = *from++;
  case 1:       *to++ = *from++;
            } while ((cnt -= 4) > 0);
}
```

(a)

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

(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

```
while (1) {

  break;

}
```

(a)

```
int swVar = 1;
while (swVar != 0) {
  switch (swVar) {
    case 1:  {
      if (1)
        swVar = 2;
      else
        swVar = 0;
      break;
    }
    case 2:  {
      swVar = 0;
      break;
    }
  }
}
```

(b)

Figure 3. Transformation of a loop with unstructured control transfer (a: original code, b: flattened code).

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 justified [11].

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 subsection. 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: $\oplus$ denotes string concatenation, while $\Rightarrow$ 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 preceeded by other statements, are eliminated to avoid visibility problems, that would result from the change in the scope of such declarations. So, the

```
while (1) {

  try {

    buf = new char[512];
    break;

  } catch (...)  {

    cerr << "exception" << endl;

  }
}
```

(a)

```
   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];
                 swVar1 = 0;
                 goto L;
               }
             }
           }
           swVar1 = 1;
         } catch (...)  {
           swVar1 = 3;
         }
         break;
       }
       case 3:  {
         cerr << "exception" << endl;
         swVar1 = 1;
         break;
       }
     }
   }
```

(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

```
levels : stack of ⟨variable, label⟩
breaks : stack of ⟨level, entry⟩
continues : stack of ⟨level, entry⟩

procedure control_flow_flattening (block)
begin
  separate variable declarations from the rest
    of block and output them before all other
    statements
  flatten_block(block)
end

procedure flatten_block (block)
begin
  while_label := unique_identifier()
  switch_variable := unique_identifier()
  entry := unique_number()
  exit := unique_number()
  ⇒ "int" ⊕ switch_variable ⊕ "=" ⊕ entry ⊕
    ";"
  ⇒ while_label ⊕ ":"
  ⇒ "while (" ⊕ switch_variable ⊕ " != " ⊕
    exit ⊕ ") {"
  ⇒ "  switch (" ⊕ switch_variable ⊕ ") {"
  push(levels, ⟨switch_variable, while_label⟩)
  transform_block(block, entry, exit)
  pop(levels)
  ⇒ "  }"
  ⇒ "}"
end
```

```
procedure transform_block (block, entry, exit)
begin
  block_parts[] := split block to parts so that
    each part is either a compound statement
    or a sequence of non-compound statements
  for each part in block_parts do
    part_exit := part is the last ? exit :
      unique_number()
    case type of part of
      block: transform_block(part, entry,
        part_exit)
      if: transform_if(part, entry, part_exit)
      switch: transform_switch(part, entry,
        part_exit)
      while: transform_while(part, entry,
        part_exit)
      do: transform_do(part, entry, part_exit)
      for: transform_for(part, entry, part_exit)
      try: transform_try(part, entry, part_exit)
      sequence: transform_sequence(part, entry,
        part_exit)
    endcase
    entry := part_exit
  endfor
end
```

Figure 5. The algorithm of control flow flattening, part one.

together with the name of the control variable is pushed to a stack (*levels*) every time a new level is created.

The procedure *transform_block*, called from the *flatten_block*, is responsible for breaking up a block to compound statements and sequences of non-compound statements, while the other *transform* procedures do the obfuscation of these block parts according to their type. The procedure *transform_if* in Figure 6 is a good example of how compound statements are obfuscated: a new case is generated in the controlling `switch` for the head of the selection, while the branches are handled by calling *transform_block* recursively on them. The procedure *transform_while* works quite similarly, except that before recursively calling *transform_block*, the case labels where the execution shall continue on a `break` or `continue` statement are pushed to two stacks, *breaks* and *continues*, respectively. Along with the case labels, the depth of the actual level of flattening, i.e., the number of entries in the *levels* stack, is pushed to both stacks as well. The same approach is used to transform `do` and `for` statements, too. The procedure *transform_switch* also uses stacking to deal with unstructured control transfer, however only the *breaks* stack is used, since `continue` statements have no effect on a `switch`.

```
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(else 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;"
  ⇒ "}"
  push(breaks, ⟨size(levels), exit⟩)
  push(continues, ⟨size(levels), entry⟩)
  transform_block(body of while_stmt,
    body_entry, entry)
  pop(breaks)
  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()
    ⇒ "  " ⊕ case_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
  test_entry := unique_number()
  body_entry := unique_number()
  ⇒ "case " ⊕ test_entry ⊕ ":  {"
  ⇒ "  if (" ⊕ predicate of do_stmt ⊕ ")"
  ⇒ "    " ⊕ switch_variable ⊕ " = " ⊕
    body_entry ⊕ ";"
  ⇒ "  else"
  ⇒ "    " ⊕ switch_variable ⊕ " = " ⊕ exit ⊕
    ";"
  ⇒ "  break;"
  ⇒ "}"
  ⇒ "case " ⊕ entry ⊕ ":  {"
  for each label in labels of do_stmt do
    ⇒ label ⊕ ":"
  endfor
  ⇒ "  " ⊕ switch_variable ⊕ " = " ⊕
    body_entry ⊕ ";"
  ⇒ "  break;"
  ⇒ "}"
  push(breaks, ⟨size(levels), exit⟩)
  push(continues, ⟨size(levels), test_entry⟩)
  transform_block(body of do_stmt, body_entry,
    test_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()
  inc_entry := unique_number()
  body_entry := unique_number()
  ⇒ "case " ⊕ entry ⊕ ":  {"
  for each label in labels of for_stmt do
    ⇒ label ⊕ ":"
  endfor
  ⇒ "  " ⊕ initialization part of for_stmt
  ⇒ "  " ⊕ switch_variable ⊕ " = " ⊕ test_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 ⊕ ":"
  endfor
  ⇒ "  try {"
  flatten_block(body of try_stmt)
  ⇒ "  }"
  for each handler in catch handlers of
    try_stmt do
    ⇒ "  catch (" ⊕ parameter of handler ⊕
      ") {"
    flatten_block(body of handler)
    ⇒ "  }"
  endfor
  ⇒ "  " ⊕ switch_variable ⊕ " = " ⊕ exit ⊕ ";"
  ⇒ "  break;"
  ⇒ "}"
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 ⊕ ":"
    endfor
    case type of stmt of
      continue:
        ⇒ levels[top(continues).level].variable ⊕
          " = " ⊕ top(continues).entry ⊕ ";"
        if top(continues).level <> size(levels)
          then
          ⇒ "goto " ⊕
            levels[top(continues).level].label ⊕
            ";"
        else
          ⇒ "break;"
        endif
      break:
        ⇒ levels[top(breaks).level].variable ⊕
          " = " ⊕ top(breaks).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

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

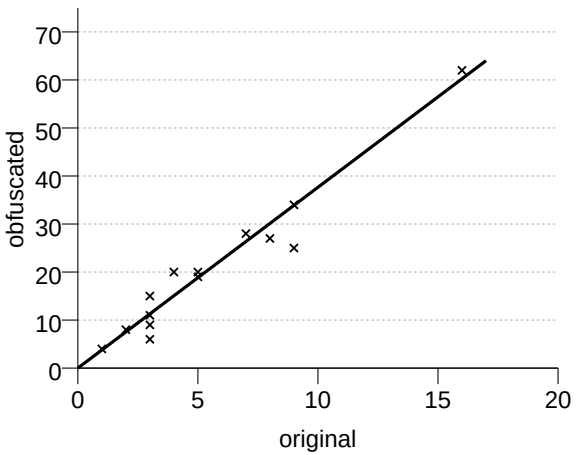


Figure 8. Relationship between the complexities of the original and the flattened code.

Table 2. The effect of control flow flattening on program size and runtime.

Function	Size (AST)	Runtime (s)
main (sumcol.cpp)	94 → 154 (1.64×)	1.53 → 1.58 (1.03×)
mmult (matrix.cpp)	61 → 162 (2.66×)	50.51 → 111.65 (2.21×)
main (almabench.cpp)	90 → 187 (2.08×)	0.12 → 0.56 (4.67×)
save_lda_model (lda-model.c)	103 → 181 (1.76×)	0.00 → 0.00 (1.00×)
new_lda_model (lda-model.c)	77 → 150 (1.95×)	0.01 → 0.01 (1.00×)
log_sum (utils.c)	39 → 77 (1.97×)	6.19 → 9.39 (1.52×)
read_data (lda-data.c)	198 → 285 (1.44×)	0.01 → 0.02 (2.22×)
matgen (linpack.cpp)	126 → 263 (2.09×)	0.72 → 1.19 (1.65×)
deep (penta.cpp)	79 → 177 (2.24×)	16.58 → 33.33 (2.01×)
gen_random (random.cpp)	18 → 30 (1.67×)	29.16 → 33.59 (1.15×)
radecdist (almabench.cpp)	92 → 127 (1.38×)	1.10 → 1.28 (1.16×)
digamma (utils.c)	81 → 92 (1.14×)	53.64 → 52.32 (0.98×)
argmax (utils.c)	34 → 91 (2.68×)	0.05 → 0.29 (5.80×)
dgefa (linpack.cpp)	494 → 810 (1.64×)	0.64 → 0.67 (1.05×)
main (moments.cpp)	105 → 197 (1.88×)	0.59 → 0.59 (1.00×)
lda_mle (lda-model.c)	101 → 195 (1.93×)	0.02 → 0.03 (1.50×)
main (nestedloop.cpp)	89 → 268 (3.01×)	96.87 → 377.48 (3.90×)
main (matrix.cpp)	112 → 166 (1.48×)	0.01 → 0.01 (1.00×)
main (sieve.cpp)	93 → 228 (2.45×)	45.39 → 98.20 (2.16×)
main (random.cpp)	56 → 93 (1.66×)	2.70 → 8.29 (3.07×)
anpm (almabench.cpp)	27 → 60 (2.22×)	0.64 → 1.24 (1.94×)
main (wc.cpp)	99 → 224 (2.26×)	39.51 → 43.08 (1.09×)
ack (ackermann.cpp)	24 → 34 (1.42×)	77.52 → 111.43 (1.44×)

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 deobfuscators. 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 occured 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 `gotos` 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

[1] **Low D.,** *Java control flow obfuscation,* Master's thesis, Department of Computer Science, University of Auckland, 1998.

[2] **Collberg C., Thomborson C. and Low D.,** Breaking abstractions and unstructuring data structures, *Proceedings of the IEEE International Conference on Computer Languages (ICCL'98), Chicago, IL, 1998.*, 28–38.

[3] **Wang C., Hill J., Knight J. and Davidson J.,** Software tamper resistance: Obstructing static analysis of programs, *Technical Report CS-2000-**12***, University of Virginia, 2000.

[4] **Chow S., Gu Y., Johnson H. and Zakharov V. A.,** An approach to the obfuscation of control-flow of sequential computer programs, *ISC '01: Proceedings of the 4th International Conference on Information Security, London, UK*, Springer Verlag, 2001, 144–155.

[5] **Zelix Pty Ltd.,** *Zelix klassmaster*, `http://www.zelix.com/klassmaster/index.html`.

[6] **yWorks GmbH.,** *yGuard*, `http://www.yworks.com/en/products_yguard_about.html`.

[7] **Baier B.,** *COBF*, `http://home.arcor.de/bernhard.baier/cobf/`.

[8] **Semantic Designs,** *Thicket family of source code obfuscators*, `http://www.semdesigns.com/Products/Obfuscators/index.html`.

[9] **Muchnick S. S.,** Approaches to control-flow analysis, *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, 1997, 172–177.

[10] **Stroustrup B.,** Expressions and statements, *The C++ programming language.,* 3rd edn., Addison-Wesley, 1997, 141.

[11] **Eckel B.,** The C in C++. *Thinking in C++.* 2nd edn., Vol. 1., Prentice Hall, 2000, 125–126.

[12] **ISO/IEC,** *International Standard – Programming languages – C++*, 2nd edn., 2003, ISO/IEC **14882**.

[13] **Ferenc R., Beszédes A., Tarkiainen M. and Gyimóthy T.,** Columbus – reverse engineering tool and schema for C++, *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, IEEE Computer Society, 2002, 172–181.

[14] **Lea K.,** *Java is faster than C++ benchmark* `http://www.kano.net/javabench`.

[15] **Netlib,** *Linpack benchmark*, `http://www.netlib.org/benchmark`.

[16] **Blei D. M.,** *LDA-C*, `http://www.cs.princeton.edu/~blei/lda-c/`.

[17] **McCabe T. J., Watson A. H.,** Software complexity, *Crosstalk, Journal of Defense Software Engineering,* **7** (1994), 5–9.

[18] **Collberg C., Thomborson C. and Low D.,** A taxonomy of obfuscating transformations, *Technical Report* **148**, Department of Computer Science, The University of Auckland, 1997.

[19] **Collberg C., Thomborson C. and Low D.,** Manufacturing cheap, resilient and stealthy opaque constructs, *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL98), San Diego, CA*, 1998, 184–196.

[20] **Sarmenta L. F. G.,** *Protecting programs from hostile environments : encrypted computation, obfuscation and other techniques,* PhD thesis, MIT, Department of Electrical Engineering and Computer Science, 1999.

[21] **Wroblewski G.,** *General method of program code obfuscation*, PhD thesis, Institute of Engineering Cybernetics, Wroclaw University of Technology, 2002.

[22] **Zhuang X., Zhang T., Lee H. H. S. and Pande S.,** Hardware assisted control flow obfuscation for embedded processors. *CASES '04: Proceedings of the 2004 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, New York, NY, USA*, ACM Press, 2004, 292–302.

[23] **Ge J., Chaudhuri S. and Tyagi A.,** Control flow based obfuscation. *DRM '05: Proceedings of the 5th ACM Workshop on Digital Rights Management, New York, NY, USA*, ACM Press, 2005, 83–92.

[24] **Lee Software**: *Smokescreen*, `http://www.leesw.com/smokescreen/obfuscation.html`

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`