Call Tree Transformation for Program Obfuscation and Copy Protection

Valery Pryamikov

Harper Security Consulting AS, Vestre Rosten 81, 7075, Tiller, Norway
valery@harper.no
http://www.harper.no/valery

Abstract. In this paper we propose a new scheme for software obfuscation and license protection that is based on an original transformation of the program's call tree. The idea is based on the observation of similarities between a program's call tree and Context Free Grammars. First, this paper proposes a practical technique for applying well studied LALR methodologies to transforming a program's call tree. Second, we suggest methods of effective binding of the transformed program to the program's installation site. Finally, we note that the given scheme provides us with a series of difficult to remove unique identifications integrally embedded into the transformed programs that could be used for software watermarking purposes.

Keywords: Software Obfuscation, Software Copy Protection, Software Watermarking.

1 Introduction

Intellectual Property Protection (IPP) related to software distribution and production is a longstanding problem. Early works in that area were mainly focused on copy protection. For examples of early work, see [9] where the author proposes some technical means of software copy protection. IPP problems related to reverse engineering and de-compilation were not considered to be as important at the time of Gosler's writing due to perceived complexity of reverse engineering of large binary-compiled programs. However, the problems of program protection against reverse engineering and de-compilation became increasingly more important and anticipated since the invention of Architecture Neutral Distribution Format (ANDF) and Virtual Execution Environment (VEE) such as Xerox-PARC's Smalltalk, Sun's Java and Microsoft's .Net. One of the major reasons for that change is that VEE/Virtual Machine (VM) Architecture Independency usually requires inclusion of rich metadata for the VEE/VM. Presence of rich metadata allows much easier de-compilation with higher than ever readability of reverse-engineered code.

In this paper we propose a new software obfuscation and copy protection scheme that is based on an original idea of program call tree transformation. We believe that the presented scheme opens a new venue for solving problems related to Software IPP.

A series of excellent theoretical and practical work in area of general Software IPP was published during the last decade. The most relevant preceding works are listed in the references section. In the remaining part of our introduction we want to emphasize the most important publications. Among these are: [5] with the first systematic classification of known obfuscating transformations, [7], [6], [16] and [18] which describes techniques that either are used or could be effectively used for augmenting the software protection framework presented in this paper.

An elegant mathematical framework studying security aspects of obfuscating transformation was introduced by [1], where authors prove the existence of classes of unobfuscatable functions. Also note a couple of later mathematical works with positive results of obfuscation [14] and [19] using the mathematical framework introduced by [1].

Context Free Grammars, LR and LALR Parsers, Call Graphs

The concept of Context Free Grammars (CFG) was first introduced by Noam Chomsky in his study of natural languages and syntactic structures. The earliest publications concerning CFG are dated to 1957-1959 with the introduction of CFG and their application to computer programming languages and formal systems. The most significant contribution to the study of CFG and parsers was done by A.V. Aho, F.L. DeRemer, J.C. Earley, D.E. Knuth and J.D. Ullman. LALR parsers were introduced by F.L. DeRemer. For further references and treatments on GFG we would refer to the reference [8].

For an account of the study of Call Graph analysis applicable to software profiling refer to works of S.L. Graham, P.B. Kessler, D. Grove and J.R. Larus. Also note [13] which suggests the use of Context Free Grammars for purposes of program profiling and introduces the notion of Whole Program Path. Other related works in the area of Program Path profiling includes publications by Melski, Ammonds, Larus, Andler and others.

Scope of Writing and Remarks

In this paper we only present an application of the algorithm to the simplest form of a call tree. Even so, the presented algorithm works well with any other type of Call Graph.

We will not discuss any details of the generation of LR(k)/LALR(1) automaton, state tables, lookup/lookahead tables but refer to related work listed in the references section.

We refer to [5] for a definition of obfuscation transformation.

For software copy protection we limit our scheme to the following:

- illegal program execution shall result in undefined random behavior;
- correct program execution shall only be guaranteed when the protected program is running in a designated environment.

The methods of identification of the program installation site, protection of the delivery path of the identification data or methods of processing of identification data are out of scope for this paper.

In this paper we will not provide any details of the application of the algorithm to exception handling; virtual methods, events and delegates; multithreading; and other advanced elements of the program control flow graphs. However we strongly believe that all mentioned programming constructs could be properly handled by an enhanced version of the presented algorithm.

2 Idea

A Context Free Grammar (**CFG**) is a formal grammar in which every production rule is of the form $A \rightarrow w$ where A is a non-terminal symbol and w is a string consisting of terminals and/or non-terminals.

CFG parsers could be implemented in several different ways, but the most usual ways are:

– a recursive descent parser – which could be thought of as a traditional procedural parser with the shape of the call tree quite closely reflecting the shape of the CFG;
– an LALR parser driver routine relying on a set of state, transition and lookahead tables with shallow and flat-shaped call tree structure;

Both are implementation of the same algorithm - «Parser», but the former implementation tends to be easier to understanding and reverse engineering than the latter.

From the other side, it's quite intuitive that CFG could be effectively used for representing a call tree; there are known works in the area of program profiling that relies on a CFG representation of a program call tree – see for example [13].

This makes us believe that we should be able to apply techniques found in LALR parsers for automatic generation of alternative representations of a program's call tree, which should provide us with an alternative representation of a program's algorithm and strong obfuscation of the source program.

In this paper we propose an obfuscation algorithm that combines several earlier ideas from C. Collberg, C. Thomborson and C. Wang[1] with original transformations of the program call tree that uses the LALR interpretation of the control flow. The algorithm also relies on obfuscation-time scrambling and runtime descrambling of LALR tables for achieving resilience against automatic de-obfuscation tools and strong copy protection. As an extra benefit, it also allows us to apply difficult to remove one-way transformations of the input alphabet, which could be useful for software watermarking purposes. The overall algorithm is:

2.1) create CFG lexer by
   a) merging all non-terminal methods of the original call tree together
      i) by merging their argument arrays;
      ii) flattening the Control Flow Graph by techniques similar to [18]; and
      iii) merging their Control Flow Graphs together;
   b) replacing the call-method instructions with return of the call-site index[2];
2.2) apply one-way transformation/(permutation) to the input alphabet from step 2.1.b) for watermarking purposes;

[1] Esp. see [5], [6], [7], [16], [17] and [18].
[2] Call-site indexes are used as an input alphabet for the CFG representation of the Call Tree.

2.3) generate an LALR driver routine that embeds terminal methods of the original call tree as CFG reduce actions;
2.4) scramble the LALR state, transition and lookahead tables by
   a) unstructuring and merging them together, and
   b) applying a set of transformations that
      i) use identification of program installation site as key material/seed; and
      ii) could be compensated/descrambled at runtime.

As a result of the application of the algorithm for transforming a source program, the original call tree becomes encoded and emulated by the LALR parse stack, which is controlled by the LALR state, transition and lookahead tables. Even minor problems during a runtime descrambling of these tables would lead to unpredictable results during a program execution. If a runtime descrambling affects a substantial part of LALR tables then it should provide a very strong copy protection because the emulated call-tree will be unusable without access to the designated installation site id[3].

Another major advantage of this algorithm is its strong obfuscation property that combines several well known obfuscation techniques due to C. Collberg, C. Thomborson and C. Wang with the original strong obfuscation of an inter-procedural control flow by flattening and reversing the actual call tree while relying on an LALR parsing for an interpretation of the logical call tree.

In cases when copy protection is considered to be a major goal, and because an LALR interpretation of the original call tree induces some performance hit to each interpreted method call; we suggest that often called, but trivial methods[4] should be excluded from a call tree CFG construction (as we will demonstrate in the following introductory example).

A strong software watermarking property comes as a convenient side-effect due to the fact that generation of an LALR parser is independent from numeric values assigned to an input alphabet as long as they stay in synch with a source CFG. We believe that the task of removing these watermarks[5] should be at least as difficult as the task of recovering the CFG (and recovering the original program call tree).

One of our design goals for the protection scheme presented in this paper was an attempt to ensure that recovering CFG/(the original call tree) from a generated LALR presentation of the call tree is indeed a difficult task; however, all questions concerning complexity of this problem is left for further study.

3 Introductory Example and Preprocessing Steps

Here we want to outline an idea of a practical implementation of the suggested scheme. For explanatory reasons we will present it on a minimal sample program. However, we believe this scheme is applicable to most real-life programs with just some adjustments/improvements. We will discuss security, performance and related considerations later in this paper.

[3] which we use as a keying material for scrambling/descrambling of LALR tables;
[4] such as property setters and getters;
[5] or switching from one permutation of an input alphabet to another permutation;

Sample Pseudo-code

During the first step we will prepare an input alphabet for our call tree CFG Parser by enumerating call sites and segments of a Control Flow Graph flattened with Wang's technique.

See Figure 1 for the pseudo-code of our sample program.

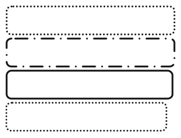
```
void Main() {
       A();
       B(message1);
}
int A()  {
       int i = C();
       while (i < D()) {
              A(); i++;
       }
       E(message2);
       return i;
}
void B(string message)  {
       F();//F will be excluded from Call-Tree CFG
       E(message);
}
int C() { //do some calculations here.
       return calculationResults;
}
int D() { //D will be excluded from Call-Tree CFG
       return --RemainingLoops;
}
void E(string message) {
       G();//G will be excluded from Call-Tree CFG
       print(message);
}
```

Fig. 1. Source Code

Preprocessing of Call Tree

Let's start with building the program call tree and preparing a set of indexes that will be used as an input alphabet for our call tree CFG parser.

3.1. Build a call tree by enumerating call sites and ignoring all methods external to the analyzed assembly (Figure 2);
3.2. Filter out trivial but often called methods[6]:
3.3. Mark all leafs (nodes without children):
3.4. Mark all joints (nodes that have at least one child):
3.5. Mark all recursive functions:
3.6. Enumerate call sites[7] (Figure 3).



[6] These methods will be treated the same way as methods external to the analyzed program.

[7] i.e. associate sequential numbers {1,2,3…} with points of calling functions on leafs (3.3) , joints (3.4) and recursive (3.5).

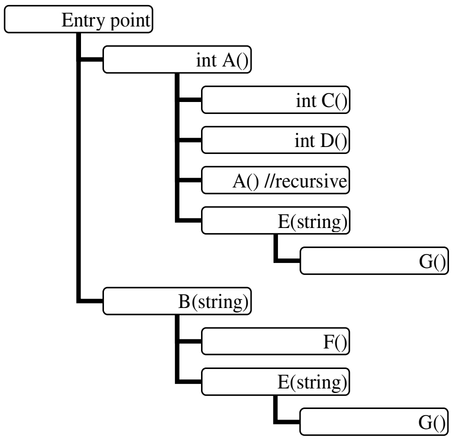


Fig. 2. Initial Call Tree

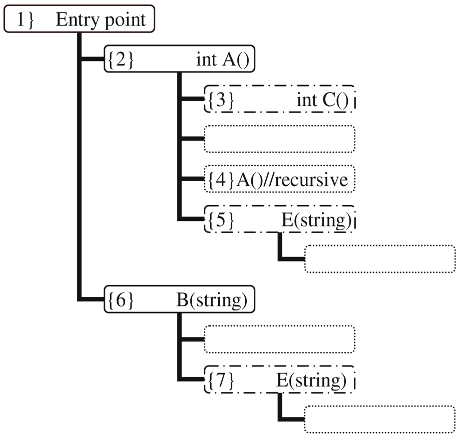


Fig. 3. Enumerated Call Sites

Application of our algorithm requires a separation of program segments surrounding the enumerated call sites. We will proceed by flattening the control flow graphs in a couple of following steps.

3.7. All loops containing at least one enumerated call site[8] should be dismantled with an algorithm such as [16]/[18] – see Figure 4.

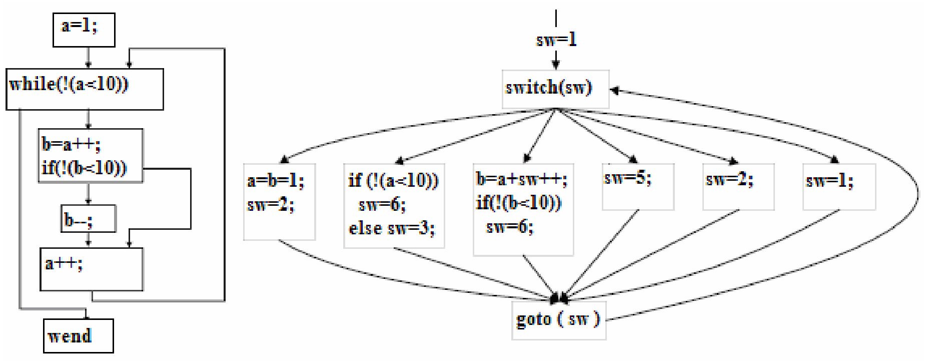


Fig. 4. Dismantling Cycles [18]

3.8. Enumerate fragments of dismantled cycles (switch labels from step 3.7);

3.9. Store the first index unused by an enumeration during steps 3.6 and 3.8 in a variable **R**. The stored value will be used for generating unique indexes required for the implementation of the CFG lexer function several steps later.

[8] Note that the source code from Figure 1 contains one cycle inside function A that requires dismantling.

4 Processing Call Tree Joints

During this stage of processing we will prepare a lexer function that will be used by our call tree CFG parser. We will only focus on call tree joints here (see step 3.4), i.e. functions **Main**, **A** and **B** of our sample code (Figure 1). The main idea is to merge their argument arrays; merge their Control Flow Graphs and replace **call** statements inside of enumerated call sites with **return** of corresponding indexes. For retaining the control flow we would need twice as many switch labels as we enumerated in the previous section for addressing Control Flow Graphs which follow the call sites. For that purpose we will use the indexes that were not used in the previous stage of processing (conveniently stored in the constant **R** during step 3.9).

Figure 5 shows an annotated version of the source code of non-terminal functions.

```
void Main() { //• entrypoint {1}
  A(); //• call site {2}
  B(message1); //• call site {6}; argB_1
}
int A() { //• return value A_ret
  int i = //• A_I_loc;
    C();//• call site {3}; C_ret;
  while (i < D()) //• loop criteria {8}
  { //• loop body {9}
    A(); //• call site {4}
    i++;
  } //• loop {10}
  E(message2); //• call site {5}; argE_1
  return i;
}
void B(string msg)//• argument argB_1
{ //• local storage argB_1_loc
  F(); //F will be excluded from Call-Tree CFG
  E(msg); //• call site {7}; argE_1
}
```

Fig. 5. Annotated Source Code

Preparing Lexer Function

4.1. Create a new function **int yylex** containing a single switch statement;
4.2. Arguments and return values of leafs and joints should be placed in a container (for example an array) which is accessible by callers of **yylex**;
4.3. Local variables that are used across any of enumerated points should be placed in the same container as in 4.2;
4.4. A reference to the container variable from the step 4.2 could be passed as a function parameter to **yylex**;
4.5. Split the source (Figure 5) on Control Flow Graph fragments (code between enumerated points);
4.6. Place all fragments from the step 4.5 into the switch statement inside **yylex**. Use the fragment indexes as case labels;

4.7. If a function inside an enumerated call-site is expecting any arguments – update correspondent arguments in the container from step 4.2 just above the call-site;

4.8. Replace the function-call inside the enumerated call-sites with return of the call site index;

4.9. Add **R** cases with the code that follows the call-sites that we replaced with return during step 4.8;

The resulting **yylex** function with the explanatory annotations is shown in Figure 6.

```
int yylex(object[] args) {
  switch (currentPosition) {
    case 1: return 2; //Main entry point; calls function A()
    case 2: return 3; //A entry point; calls function C()
    case R+2:args[argB_1]=message1; // argument to B();
      return 6; //call function B();
    case R+3:args[A_I_loc]=args[C_ret];
      // i ← return from C();
      goto case 8; //go to loop criteria;
    case 4: goto case 2; //A - recursive call; goto A's entry point
    case R+4:goto case R+9; //A returns; continue loop.
    case R+5:args[A_ret]=args[A_I_loc];
      //return from E(); update A's retval
      break; //exit A;
    case 6:args[argB_1_loc]=args[argB_1];
      //B's entry point; argument → local storage
      F(); args[argE_1]=args[argB_1_loc]; //arg. to E();
      return 7; //calls E();
    case R+6:break;//return from B(); exit Main
    case R+7:break;//return from E(); exit B();
    case 8: //loop condition
      if (args[A_I_loc] < D())
        goto case 9; //go to loop body;
      else
        goto case R+10;//exit loop
    case 9: return 4; //calls A() - recursive;
    case R+9:args[A_I_loc]++;//increment loop variable
      goto case 10; //go to loop;
    case 10: goto case 8; // go to loop criteria;
    case R+10:args[argE_1]=message2; //argument to E();
      return 5; //calls E()
  }
  return 0; //(end-of-branch/reduce);
}
```

Fig. 6. The annotated pseudo-code of the yylex function

5 CFG and LALR Transformation of Call Tree

In this section we will construct a Context Free Grammar over a set of terminal symbols $V_T \subseteq T$, where T is a set of lexical tokens/values returned by the **yylex** function which we built in the previous section. Our CFG will be representing the original call tree. After that we will build an LALR parser function that will be emulating the original call tree by means of an internal parse stack. Finally, we will build first the version of the obfuscated program consisting of our LALR parser and the **yylex** from the previous section.

We will not elaborate on algorithms used by LALR parser generators or LALR parser driver routines, but instead we will refer to work of F. L. DeRemer, S. C. Johnson, R. Corbett and the related literature listed in the reference section (see [8]), as well as source codes of open source implementations of LALR parsers[9].

The LALR parser driver routine used in our scheme should ensure that updates of the internal parse stack position will be correctly reflected in the **currentPosition** variable that we used in the **yylex** function as a switch control variable (see Figure 6).

Additionally, the leaf functions **C** and **E** (Figure 1) will be inlined in the reduce actions of our CFG.

In Figure 7 is a raw sketch of a grammar definition of our call tree.

```
Main: BranchA BranchB ⊥;
A_1: A C      {inline C;};
A_2: A_1 A    {/*recursive A()*/;}
| A_2 A       {/*recursive A()*/;}
| A_2 ⊥;
A_3: A_2 E    {inline E;};
BranchA: A_3 ⊥;
B_1: B E      {inline E;};
BranchB: B_1 ⊥;
```

Fig. 7. Call Tree Grammar Definition

5.1. Use a grammar definition to generate an LALR parser driver routine that is also updating the **currentPosition** variable of the **yylex** function;

5.2. The leaf functions (**C** and **E**) should be inlined as reduce-actions of the LALR Parse function by using any standard inlining method. They also require the use of the container from the step 4.2 for retrieving parameters and storing return values.

Figure 8 shows the relevant fragments of the **yyparse** function that illustrates the call of **yylex** and the inlining of reduce-actions. The rest of logic of the LALR parser driver routine is omitted from Figure 8.

[9] Such as YACC/BYACC and BISON.

```
int yyparse() {
  object[] yyargs;
  //...intialize yyargs here
  ...
  //...LALR logic here
  pcyytoken=yylex(yyargs);
  //...LALR logic here
  ...
   switch (m) { /*actions associated with grammar rules*/
      case 3: { //do some calculations here.
       yyargs[C_ret] = calculationResults;
       //return calculations result
       } break;
      case 5: //falls through
      case 7: {
       G(); //G is often called method which we excluded from CFG
       print(yyargs[argE_1]); //prints message sent in parameter
     } break;
      ...    }
   goto enstack; }
```

Fig. 8. Fragments of LALR Parse pseudo-code

Now we are ready to create the first obfuscated version of our program that uses the LALR call tree obfuscation technique.

5.3. Our LALR obfuscated program will be created by putting together:

a. **yylex** (generated during steps 4.1—4.9);
b. **yyparse** (generated during steps 5.1—5.2), which calls **yylex** (see a. above);
c. the entry point function which sets the **currentPosition** to 1 and calls **yyparse** (see b. above).

If we take another look at our transformation, it essentially means that we have reversed and flattened the call tree, so that:

– all leaf functions from the lowest level of the call tree are now moved into a single **yyparse** function at the top of the modified call tree;
– all other functions, that were directly or indirectly calling the former leaf functions (see above), are now moved to a single leaf function **yylex** (regardless of their original call tree position).

The functionality of the original program is preserved by moving the original call tree into the parse stack of an LALR parser.

The LALR Parse stack is controlled by the interpretation of the state, lookup and lookahead tables. Figure 9 shows a sample of the LALR tables generated by YACC.

```
const int yyact[] = {
       5,       0,       0,       4,       8,       3,      17,       8,
       7,       9,       6,       0,       9,       8,       7,       1,
       6,       0,       9,      10,      11,      12,      13,      14,
      15,      16,       0,       0,       0,       0,       0,       0,
...
};
const int yypact[] = {
     -40,     -29,   -4096,     -40,     -40,     -40,     -40,     -40,
     -40,     -40,   -4096,   -4096,     -35,     -38,     -38,   -4096,
   -4096,   -4096,
};
const int yypgo[] = {
       0,      15,
};
const int yyr1[] = {
       0,       1,       1,       1,       1,       1,       1,       1,
       1,
};
const int yyr2[] = {
       0,       1,       2,       2,       3,       3,       3,       3,
       3,
};
const int yychk[] = {
   -4096,      -1,     257,      45,      43,      40,      45,      43,
      42,      47,      -1,      -1,      -1,      -1,      -1,      -1,
      -1,      41,
};
```

Fig. 9. The state, lookup and lookahead LALR tables generated by YACC

6 Protecting LALR Tables and Adding Copy Protection

The main problem with the LALR tables shown in Figure 9 is that their well defined structure could be used for recovering the source CFG with the help of the specially designed programs. Fortunately, we believe that there are ways of protecting LALR tables from such a threat. In fact, there are known techniques of arrays obfuscations that could be used for such purpose, as for example Array manipulations and String Encoding transformations by C. Collberg and C. Thomborson [7].

Additionally, if we derive a transformation key[10] from some unique installation site ID, then it will also provide us with a very efficient copy protection, because if the LALR tables only could be recovered in the presence of an unique installation site ID, then any attempt to run such a program on a different installation site would lead to distorted LALR tables, a corrupt call tree and completely unpredictable results. Unfortunately, a simple derivation of a symmetric encryption key from an installation side ID; encrypting LALR tables during obfuscation time and decrypting them during runtime could only provide a marginal protection (if any at all). The latter is due to the simple fact that when a complete LALR table structure is decrypted in a process memory it immediately becomes a subject to various attacks including simple dumping of decrypted LALR tables and running analysis of the memory dump.

Therefore, we would require a complex set of counter-measures that includes the obfuscation of the tables structure; use of various table access obfuscation techniques,

[10] We will use it for scrambling of the LALR tables.

such as added indirection layers, alias-tables and alias rotations; ensuring that only small, immediately required parts of LALR tables be descrambled at any given moment in time, while the rest of this structure should be kept protected. Another important factor to ensure is that it should be difficult to distinguish scrambled parts of the tables from the descrambled parts. Reasonable candidates of our protection framework could be based on the ideas from [14] where authors show how to obfuscate a complex access control functionality, and demonstrate strong access obfuscation properties of regular expressions and related functions. Another protection measure could be modeled on unstructured LALR tables and/or alias-tables like expander graphs and using bytes of a cryptographic hash of an installation site id for choosing walk edges. The expanding property of the graph implies (via a non-trivial proof) that the vertices along random walks on an expander have surprisingly strong random properties [2]. We can also XOR bytes along the walking path with another result of cryptographic hash as a part of a scrambling and descrambling processes.

In other words our goals are:

- to scramble LALR tables at obfuscation-time by using some function dependent on an installation site id;
- a periodic descramble of required parts of these tables at runtime;
- ensure that descrambling of these tables without having access to a corresponding installation site id is a difficult task, while the runtime descrambling of these tables only has an insignificant impact on performance.

Draft Description

6.1. We need to start with expanding and unstructuring tables, e.g. merging them into a single array and applying an initial permutation that could be matched by one or more layers of an added indirection with a help of alias tables (or similar) [7].

6.2. A similar set of transformations could be applied to both LALR tables and alias tables.

6.3. Addressing subsets of these tables should expose strong access obfuscation properties and pseudo-random properties. We can:

   - model unstructured tables as an expander graph and use bits from PRF(id || hour)[11] for choosing the walk edges;
   - use a regular expression over random variables that are mappings of (possibly unadjusted) bytes of PRF(id || hour), where id should be some unique identification of a program installation site and PRF could be a cryptographic hash function (let say SHA-1).

6.4. Addressed subsets could be used with different transformations that are efficiently computed at runtime, such as:

   - removing/inserting addressed subsets;
   - XOR-ing two (or more) addressed subsets together;

[11] Implementation of the algorithm will use a cryptographic hash function as a practical substitution of PRF.

- Using a modular arithmetic with bytes of addressed subsets at runtime and an inverse modular arithmetic at obfuscation-time;
- other suitable transformations.

6.5. Transformations listed in the previous step could be executed by a function running on a separate execution thread and also executed at the startup of a protected program.

Processing Results

- If descrambling of relevant parts of LALR tables was incorrect, it will severely affect the ability of the generated LALR parser to depict a correct shape of the original call tree. This will lead to unpredictable results of the program execution.
- Correct descrambling of the relevant parts of the LALR tables and the alias-rotation tables will only be guaranteed in the presence of the correct installation site id.
- We believe that LALR tables obfuscated and scrambled this way will provide an efficient protection against attempts of recovering the source CFG.

7 Adding Watermarks

If we look back at our choice of the input alphabet (steps 3.6—4.9) it's clear that we only require unique indexes and our choice of sequential numbers is arbitrary, supported only by convenience and explanatory reasons. A generation of the LALR parser is independent from the numeric indexes assigned to the input alphabet as long as indexes stay in synch with the source CFG. We can add an extra step with a permutation of the input alphabet before we generate the **yylex** and the **yyparse**.

Here is a draft description of the algorithm:

7.1. put all indexes (call sites, dismantled cycles fragments and all previous indexes incremented by **R**) into an array or a table;
7.2. the table from the previous step (7.1) could be augmented with aliases (e.g. 23 aliases for each index so that we can cycle indexes once per hour);
7.3. generate a random encryption key and use it to encrypt the table;
7.4. in cases when a permuted input alphabet is intended for watermarking purposes – store the encryption key generated in the previous step together with the original tables from step 7.1. Otherwise, if a post-identification of the watermark is not required, the encryption key generated in step 7.3 simply could be destroyed;
7.5. map each number in the original table to the corresponding position in the encrypted table and use the numbers from the encrypted table in the body of the protected program;
7.6. cases of the switch statement could be sorted in an ascending or descending order or randomized;
7.7. a return of indexes from the **yyparse** function could be replaced with a return of elements of an indexed (rotated) collection of input alphabet aliases (step 7.2);
7.8. excessive cases with a slightly modified/buggy code could be randomly placed in the switch body of the **yylex** and **yyparse** functions;

7.9. excessive cases could be combined with aliases to behave as a buggy code before they are selected by the alias scheduler;
7.10. the alias scheduler could be implemented by the same routine that descrambles parts of the LALR tables (step 6.5).

Processing Results

– The permuted input alphabet becomes an integral part of the **yylex** and **yyparse** functions as well as the state, lookup and lookahead tables generated by our obfuscating transformation.
– We believe that replacing a whole permuted input alphabet with another permutation is as a difficult task as a task of recovering a source CFG.
– Partial replacements of just a few symbols from an input alphabet could be easily matched by using “tree proximity” measurements for a detection of watermarks.

8 Performance Impact and Final Remarks

It is clear that the call tree emulated by the parse stack of the **yyparse** routine would not provide as a good performance as a direct function call. However, we believe that by adjusting how many often-called-but-trivial methods shall be excluded from the transformation, it is possible to achieve a strong-enough protection without affecting an overall performance of the protected program. The latter is especially true when concerning interactive applications, where a performance impact could be made completely unnoticeable. Very preliminary results of our tests show that occasional fluctuations of an execution environment (such as window scroll, thread context switch, processor speed steps, auxiliary inputs, such as mouse movement and others) could have a greater effect on the performance than the emulation of a call tree by the parse stack of the **yyparse** function. We have run our tests on three different computer configurations – Intel Centrino with Pentium M 1.6 MHz/1 GB; Dual Pentium 4 3.2 MHz / 2 GB and Intel Pentium 4 Celeron 2.2 MHz/512MB; all running Windows XP. All three computers used for tests didn’t show a performance difference between the original and the transformed programs. Figure 10 shows results of several runs of the test program on the Pentium M computer. The test program source code could be found on the author’s web page [20].

`Source.exe /noninteractive`	`Transformed.exe /noninteractive`
`completed in  547.90 ms.`	`completed in  558.25 ms.`
`completed in  557.24 ms.`	`completed in  555.48 ms.`
`completed in  550.65 ms.`	`completed in  554.38 ms.`
`completed in  554.61 ms.`	`completed in  549.05 ms.`

Fig. 10. Sample results of performance test on the Pentium M 1.6/1GB computer

Even so the results of our test may appear surprising, there is a simple explanation of the results. The percentage of a processing time that a normal program spends on transforming a control flow between different functions is usually very small when compared to the processing time used for actual calculations and/or calling external functions. For purposes of a closer emulation of a normal processing behavior of interactive programs, we avoided adding calls to an LALR emulated functions from tight loops. Therefore an LALR emulation of the call tree has only affected a very small (by percentage of execution time) part of the program and the absolute performance impact happened to be lesser than effects of occasional fluctuations of the execution environment. We want to note that the results of our performance test are very preliminary for drawing conclusions about a performance impact on real life programs. A bigger size of a call tree and a bigger size of LALR tables are the two most obvious factors that could affect a performance. We are very optimistic in our expectations, however an additional study is required for presenting more accurate estimates of a performance impact and providing recommendations for minimizing it for different classes of real life applications.

Finally, we want to note that even a simple examination of the source code of our sample test application appears to be quite reassuring about obfuscation properties of the suggested transformation. Additionally, other known methods of a program and data obfuscation could be effectively used in a combination with our scheme for augmenting resilience and potency of the obfuscation transformation proposed in this paper.

Acknowledgements. The author would like to thank Dr. Christian Collberg for his invaluable help on this paper.

References

1. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Sali Vadhan, Ke Yang. On the (im)possibility of obfuscating programs. In Proceedings of CRYPTO 2001.
2. N. Biggs. Algebraic graph theory, 2nd ed., Cambridge University Press, 1994. ISBN 0-521-45897-8.
3. Christian Collberg, Clark Thomborson. Watermarking, tamper-proofing and obfuscation – tools for software protection. Technical Report TR00-03, The Department of Computer Science, University of Arizona, February 2000.
4. Christian Collberg, Clark Thomborson. Watermarking, tamper-proofing and obfuscation – tools for software protection. IEEE Transactions on software engineering, vol.28, No.8, August 2002.
5. Christian Collberg, Clark Thomborson, Douglas Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, University of Auckland. July 1997.
6. Christian Collberg, Clark Thomborson, Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. Principles of Programming Languages 1998, POPL'98, January 1988.
7. Christian Collberg, Clark Thomborson, Douglas Low. Breaking Abstractions and Unstructuring Data Structures. IEEE International Conference on Computer Languages, May 1998.
8. A.V. Aho, R. Sethi and J.D. Ullman, Compilers: Principles, Techniques and Tools. Addison Wesley, 1986, ISBN 0201100886.

9. James R. Gosler. Software Protection: Myth or Reality? Sandia National Laboratory. Advances in Cryptology – CRYPTO'85. 1985.
10. D. Grove, G. DeFouw, J. Dean, C. Chambers. Call Graph Construction in Object-Oriented Languages. Proceedings of OOPSLA '97. pp. 108-124, 1997
11. Horwitz, S., Precise flow-insensitive may-alias analysis is NP-Hard, ACM Transactions on Programming Languages and Systems, Vol 19. No.1, pp 1-6. 1997.
12. W. Landi, Undecidability of static analysis. ACM Lett. Program. Lang. Syst. 1, 4, 323-337. 1992
13. James R. Larus, Whole Program Paths, Proceedings of the SIGPLAN '99 Conference on Programming Languages Design and Implementation (PLDI 99), May 1999, Atlanta Georgia.
14. Benjamin Lynn, Manoj Prabhakaran, Amit Sahai. Positive Results and Techniques for Obfuscation. In Proceedings of Eurocrypt 2004.
15. G. Ramalingam, The undecidability of aliasing, ACM Trans. Program. Lang. Syst. 16, 5,1467-1471, 1994
16. Chenxi Wang. A Security Architecture for Survivability Mechanisms. PhD Dissertation, Department of Computer Science, University of Virginia, October 2000.
17. Chenxi Wang, Jonathan Hill, John Knight, Jack Davidson. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report CS-2000-12, Department of Computer Science, University of Virginia. May 2000.
18. Chenxi Wang, Jonathan Hill, John Knight, Jack Davidson. Protection of Software-based Survivability Mechanisms. International Conference of Dependable Systems and Networks. July 2001.
19. Hoeteck Wee. On Obfuscating Point Functions. Computer Science Division University of California, Berkeley. Jan 2005
20. Valery Pryamikov. Call Tree Transformation. Test Program – Source Code http://www.harper.no/valery/CallTreeTransformation