On the Effectiveness of Source Code Transformations for Binary Obfuscation

Matias Madou Bertrand Anckaert Bruno De Bus Koen De Bosschere
Department of Electronics and Information Systems
Ghent University
St.-Pietersnieuwstraat 41
B-9000 Ghent, Belgium.
{mmadou,banckaer,bdebus,kdb}@elis.ugent.be

Jan Cappaert Bart Preneel
Department of Electrical Engineering
Katholieke Universiteit of Leuven
Kasteelpark Arenberg 10
B-3001 Heverlee, Belgium.
{jan.cappaert,bart.preneel}@esat.kuleuven.be

Abstract

Obfuscation is gaining momentum as a protection mechanism for the intellectual property contained within or encapsulated by software. Usually, one of the following three directions is followed: source code obfuscation is achieved through source code transformations, Java bytecode obfuscation through transformations on the bytecode, and binary obfuscation through binary rewriting. In this paper, we study the effectiveness of source code transformations for binary obfuscation. The transformations applied by several existing source code obfuscators are empirically shown to have no impact on the stripped binary after compilation. Subsequently, we study which source code transformations are robust enough to percolate through the compiler into the binary.

Keywords: Source and Binary Obfuscation, Software Security, Intellectual Property, Program Transformations

1 Introduction

While copyright laws cover most of the intellectual property contained within or encapsulated by software, economic realities teach that these laws are often hard to enforce. As a result, many software providers revert to technical means of protection, instead of legal means.

Many different technical security solutions have been devised, some of which provide a high level of security, at the cost of limited applicability. For example, when neither network bandwidth nor latency is an issue, software can be run from a remote server. That way, the user will not gain physical access to the program. If the end user can be convinced to use tamper resistant hardware, the program can be encrypted before shipping and decrypted and executed entirely in hardware.

In many situations however, we cannot change the software distribution model or user hardware. Obfuscation can provide a certain level of protection in these situations. While a competent attacker will always be able to reverse engineer a program, given enough time and perseverance, obfuscation could make the attack economically inviable, i.e. the cost of the attack could outweigh the benefits of a successful attack.

Depending on the format in which software will be distributed, different types of obfuscation can be applied. When the source code of a program needs to be distributed, source code obfuscation can be applied. Likewise, bytecode obfuscation can protect software which will be distributed as Java bytecode, and binary obfuscation can protect software which is distributed as native code.

Typically, there is a one-to-one mapping between the type of transformations that are applied and the type of obfuscation that is pursued: source code obfuscation is attained through source code transformations, bytecode obfuscation through transformations on the bytecode and binary obfuscation through binary rewriting. This paper studies how well binary obfuscation can be achieved through source code transformations.

Source code transformations for binary obfuscation have a number of potential advantages over binary transformations: (i) source code contains more high-level information about the software, (ii) source code is architecture-independent, and (iii) code added for obfuscation purposes may blend in better with the existing code. Local binary

transformations can often be detected because the characteristics of the modified code differ from the characteristics of the unmodified code.

On the other hand, source code transformations have a number of potential disadvantages compared to binary transformations: (i) the transformations may be undone by the compiler, (ii) low-level information, such as addresses, is not yet available, and (iii) usually, there is no global view of the entire program, as source code transformations will typically be performed per compilation unit.

Bytecode transformations could be situated in between source code and binary transformations. As bytecode contains the low-level instructions for the virtual machine and still contains virtually all source level information, this facilitates the transformations. On the other hand, the degree of obfuscation that can be applied to Java bytecode is limited, as it should still be able to pass the bytecode verifier.

Given the potential advantages of source code transformations, it is interesting to study how useful source code transformations are for binary obfuscation and to study how the potential disadvantages can be avoided. Furthermore, applying source code transformations for binary obfuscation might be able to exploit the advantage of having source level information with less restrictions on the degree of obfuscation that can be present in the low-level representation.

The remainder of this paper is structured as follows: Section 2 discusses related work and positions this research within the domain of software obfuscation. Next, two frameworks for source-to-source transformation are discussed in Section 3. These frameworks are used in Section 4 to study which types of transformations survive a pass through a compiler and stripping tool. Finally, conclusions and future work are drawn in Section 5.

2 Related Work

The goal of software obfuscation is to prevent an attacker from understanding the software to which he has access. This attacker could have access to the software in different forms. He could, e.g., have access to the source code, to a native executable or to Java bytecode. In this section, we will divide the field of software obfuscation into categories according to the form from which an attacker is assumed to depart to obtain program understanding.

2.1 Source Code Obfuscation

Source code obfuscation consists of any technique that is targeted at making the source code less intelligible. In order to be useful, the resulting source code should still compile and result in a functionally equivalent program.

Source code obfuscation can thus take place at any point in the software development cycle before the source code is abandoned for a more low-level representation. Programming software in a bad way is a possible way of making more complicated source code. In practice, this way of writing software is rarely an option. Therefore, obfuscation is best added automatically at a point where human interaction is no longer required. For source code, the only remaining option is therefore to automatically transform the programmer's source code into more complex, functionally equivalent source code.

A notable exception, where obfuscation is not added automatically, is the International Obfuscated C Code Contest[1]. In this contest, laboriously hand-crafted programs compete for the title of the most obfuscated program. While this has lead to dazzling, visually very appealing, illustrations of how far source code obfuscation could go, its labor-intensiveness makes it inviable for large projects where maintainability and time-to-market are important factors.

Not surprisingly, the process of obfuscating source code has been automated through source code transformations which guarantee functional equivalence and make the code more complex for a human observer. The commonly applied transformations include: (i) replacing symbol names with non-meaningful ones, (ii) substitution of constant values with arithmetic expressions, (iii) removing source code formatting, and (iv) exploiting the preprocessor.

Tools such as the *SD Obfuscator for C(GCC3)* by SD[2], the source code obfuscator *CXX-Obfus* by Stunnix[3] and the freeware *C/C++ Sourcecode Obfuscator (COBF)* by Bernhard Baier[4] all apply one or more of these transformations. The output of these tools shows that program understanding through simple visual inspection is indeed seriously hampered. An example is given in Figure 1.

In Section 4.1, we study the effect of these transformations on a compiled, stripped binary. As will be discussed, these transformations do not change the resulting binary. This may very well have been a deliberate design decision, as it also guarantees that performance will not be adversely affected by these transformations.

2.2 Binary Obfuscation

Binary obfuscation aims at making the binary representation of software more difficult to understand. Stripped, native code is inherently harder to understand than source code, as binary code is more low-level and contains fewer abstractions than source code. However, it is still not impossible and a number of techniques have been devised to further complicate the understanding of binary code.

Binary obfuscation can be obtained through binary rewriting [22]. The difficulties in binary rewriting can however seriously limit the transformations that can safely be applied to the binary. Therefore, most applications of binary rewriting use extra information (relocation information, boundaries between code and data, ...) to guarantee correctness [12, 14].

Binary rewriting has a number of advantages over source code rewriting: transformations which require information about the exact addresses or the assembly instructions cannot be applied on source code as this information is not yet available. For example, the use of code addresses as data or

[1] http://www.ioccc.org/

[2] http://www.semanticdesigns.com/Products/Obfuscators/CObfuscator.html

[3] http://www.stunnix.com/

[4] http://home.arcor.de/bernhard.baier/cobf/

```
int my_output()
{
  int count;
  for (count = 0; count < MAX_INDEX; ++count)
  printf("Hello %d!\n", count);
}
```

(a)

```
#define a int
#define b printf
#define c for
a l47(){a l118;c(l118=0;l118<0x664+196-0x71e;++l118)
b("\x48\x65\x6c\x6c\x6f\x20\x25\x64\x21\n",l118);}
```

(b)

Figure 1. Example source code before (a) and after (b) obfuscation

overlapping instructions cannot be specified in source code. Furthermore, as this is the final step in the software development cycle, no subsequent step will cancel out the effect of the applied transformations. A final advantage that binary rewriting may have over source code rewriting is that it is typically performed on the entire program, while source code rewriting is usually done per compilation unit.

Binary rewriting is however limited by the absence of high-level information, such as type information, which complicates the transformation of datastructures. Furthermore, the location of the changed or added code is sometimes easily detected as this code might blend in poorly with the original code [1]. For example, code added after register assignment often needs to free registers to perform computations and this could lead to unusual register spills. If the transformations would be applied on the source code, register assignment would be performed in the same way as for the original code. Likewise, the original code could contain compiler idioms which are not or no longer present in the added or changed code. An additional disadvantage of binary rewriting is its architecture-dependence.

When the transformation can be specified at the source code level, a number of these disadvantages can be overcome. For as far as we are aware, the only publication which deals with source code transformations is due to Wang *et al.* [20]. She has suggested a number of control flattening transformations, which are applied on the source code and the effect of these transformations is evaluated at the binary level.

Note that we were unable to evaluate the source code transformations applied by the commercial software protection tool by Cloakware[5], because we did not receive a version for research purposes. According to the website, the *Cloakware Transcoder is a command line utility that transforms source code into mathematically modified source. When compiled with commercial, off-the-shelf compilers, transcoded source results in object code that is functionally identical to the original but resistant to reverse engineering and tampering attacks.*

2.3 Java Bytecode Obfuscation

Java bytecode obfuscation is, in a way, similar to binary obfuscation: the binary representation for the Java Virtual Machine is obfuscated. On the other hand, it can be compared to source code obfuscation, because it contains virtually all source code information. Because bytecode contains that much information, it is very susceptible to reverse engineering. As a result, a considerable amount of research has been done on the obfuscation of Java bytecode [4, 9, 11, 13].

The same holds for the Common Intermediate Language (CIL) of the .NET Framework, but because most of the work on the obfuscation of CIL is proprietary, e.g., Dotfuscator by PreEmptive Solutions[6], we will focus on the obfuscation of Java bytecode.

Java bytecode contains extensive information, mainly to enable the bytecode verifier to verify the reliability of the code. This verifier needs to check if the code (i) does not forge pointers, (ii) does not violate access restrictions, and (iii) accesses objects as what they are. To this end the bytecode verifier traverses the bytecodes, constructs the type state information, and verifies the types of the parameters to all the bytecode instructions. This enables safe execution of untrusted bytecode. Clearly, to ensure this, a number of restrictions are imposed on what bytecode may do. For example, (i) once code has been loaded into the Java interpreter, it cannot modify itself, (ii) there can be no operand stack overflows or underflows, (iii) the types of the parameters of all bytecode instructions should always be verifiably correct, and (iv) object field accesses have to be legal. None of these restrictions applies to binary code.

On the one hand, this limits the extent to which Java bytecode can be obfuscated: the bytecode verifier should still be able to prove the reliability of the code no matter how high the degree of obfuscation. Likewise, because this

[5] http://www.cloakware.com/

[6] http://www.preemptive.com/products/dotfuscator/

type of code still contains a lot of high-level information, more obfuscating transformations can be applied than on binary code. For example, type information is available and thus transformations on data are in this case easier applicable than on a binary program.

3 Source Code Transformations

Transformations on source code can be performed on different intermediate representations. Similarly to the many different intermediate representations used by compilers to transform source code into binary code, different representations of the source code can be used to transform source code into source code. Different representations are typically useful in different contexts. Unfortunately, none of them is ideal for all applications, otherwise we would notice a convergence towards this optimal representation.

While many different intermediate representations have been devised for the use in compilers, they cannot all be used for source-to-source transformations, as some of them are too low-level to be easily transformed back to source code. For this paper, we have looked at a number of different source-to-source transformation frameworks, based on different intermediate representations, of which we have selected two for deeper inspection[7]. These frameworks are TXL (Turing eXtender Language) [7] and SUIF (Stanford University Internal Format) [10]. We will now continue with a discussion of these two frameworks.

3.1 TXL

TXL [8] is a programming language and rapid prototyping system specifically designed for implementing source transformation tasks. It was originally designed in 1985 and has since been used in a wide range of practical applications. It is developed and maintained by the Software Technology Laboratory of Queen's University, Canada. The TXL paradigm consists of parsing the input text into a parse tree according to a grammar, transforming the parse tree to create an output parse tree, and unparsing this new parse tree to create the output text. A parse tree is a high-level representation, very close to the source code and can thus easily be transformed back into source code.

In TXL, transformations are specified using a set of transformation rules. Each transformation rule specifies a target type to be transformed, a pattern, and a replacement. When a pattern is matched, variable names are bound so that they can be used in the replacement to copy their bound instance into the result. The pattern and replacement are specified as augmented source text. As such, the specification of transformations is very intuitive.

3.2 SUIF

The SUIF compiler system is a free infrastructure designed to support collaborative research in optimizing and parallelizing compilers. It was originally designed in 1991 and a first public version was released in 1994. It was developed by the Stanford Compiler Group of Stanford University, California, USA. Unfortunately, the last release dates back to 1999 and SUIF is no longer maintained.

The kernel of SUIF, a.o., defines the intermediate representation and provides functions to access and manipulate the intermediate representation. The intermediate representation is a mixed-level representation. Besides low-level operations, it includes three high-level constructs: loops, conditional statements, and array accesses. The symbol tables in a SUIF program hold detailed symbol and type information. This additional information allows us to translate the intermediate representation back into legal C code.

Functions are provided to access and manipulate the intermediate representation, which hide the low-level implementation details. However, the user is still very aware of the internal structure of the intermediate representation and needs to specify the transformations in a very procedural way. As such, it is less intuitive than TXL. On the other hand, we found it to be more powerful, as we will discuss in Section 4.

4 Source Code Transformations for Binary Obfuscation

In this section, we will study the impact of source code transformations on the stripped binary obtained after a pass through the compiler and a stripping tool. Stripping discards symbols from object files and is a first form of obfuscation which should always be applied when distributing a binary program if a minimal level of protection against program understanding is required.

We will follow the classification of obfuscating transformations introduced by Collberg *et al.* [6]. This classification is based upon the transformation target and distinguishes between layout obfuscation, data obfuscation, and control obfuscation. We do not consider preventive obfuscation here, as it is typically applied on a lower level than source code. While most of the transformations in the aforementioned paper are described in a generic way, they have been designed and implemented against the background of Java bytecode. To the best of our knowledge, unless stated otherwise, this is the first time they have been applied on source code for binary obfuscation.

4.1 Layout Obfuscation

In the context of C code obfuscation, we could exploit the preprocessor to make the code unreadable and we could scramble identifiers, change formatting and remove comments. As can be seen in Section 2.1, these types of obfuscation are already commonly used for source code obfuscation. However, not a single of these layout obfuscation transformations survives the compilation process. The resulting stripped binaries are identical to those where no layout obfuscation was applied. This could be expected, as preprocessing is already performed automatically, comments and formatting are not part of the binary and stripping already scrambles identifiers. We would like to state

[7]Note that we were unable to take commercial program transformation systems such as DMS [3] into account.

clearly that layout obfuscation is useful when delivering IP in source form to customers, for some classes of customers, and not for others.

Remark that changing library function names will end up in the final program. The drawback is that all programs calling a library function should also be changed and this makes the use of such obfuscation transformation very limited. As layout obfuscations does (in general) not survive the compiler process, we focused on the remaining obfuscation transformations: control flow obfuscations and data obfuscations.

4.2 Control Flow Obfuscation

Control flow obfuscation consists of applying transformations in order to hide the control flow of a program. Well-known examples of control flow transformations are *control flow flattening* [19] and the insertion of *opaque predicates* [2, 5]. For this paper, we inserted opaque predicates, flattened a program through a control flow flattening algorithm, and turned a reducible control flow graph into a non-reducible one.

Opaque Predicates An opaque predicate is a predicate whose evaluation is known at obfuscation time, but whose evaluation is hard to deduce afterwards [5]. They were first described by Collberg *et al.* [6]. Opaque predicates have since been used for obfuscation [6], watermarking [21], tamper-proofing [18], and to protect mobile agents [16]. A true opaque predicates P^T always evaluates to *true*, while a false opaque predicates P^F always evaluates respectively to *false*. Many true and false opaque predicates can be found in literature; for example the true opaque predicate $\forall x \in \mathbb{Z}^+ : 9|10^x + 3.4^{x+2} + 5$ and the false opaque predicate $\forall x \in \mathbb{Z} : 7 \not| x^2 + 1$ [17].

Clearly, trivial opaque predicates, such as $4 \equiv 0 \pmod 2$ will not survive the optimization step. This could be remedied by assigning the constants to new global variables. Then, as the compiler cannot assume that they are constant, they will survive the optimization phase of the compiler. For more complicated opaque predicates, it is difficult to know which property has been used and therefore, they are hard to remove.

The insertion of a non-trivial opaque predicate based on variables requires type information of the variables, because the programmer has to make sure overflow will not occur during the computation of the opaque predicate. In theory, $3|x(x^2-1)$ is always *true*, but in practice this computation will be executed in, e.g., a 32-bit environment which is vulnerable to overflow. When the type of x is known, an appropriate domain could be taken to make sure no overflow will occur. For example, in a 32-bit environment the opaque predicate $\forall x \in [0, 1024] : 3|x(x^2-1)$ will never result in overflow. SUIF can easily determine the type of the variable used, while in TXL, this is more difficult.

Control Flow Flattening Control flow flattening is introduced in the PhD dissertation of Wang [19] and aims to obscure the control flow of a program by "flattening" the control flow graph. As can be seen in Figure 2, the default algorithm transforms a function into a functionally equivalent one where all direct control flow edges between code are changed into control flow edges towards a single redirection block.

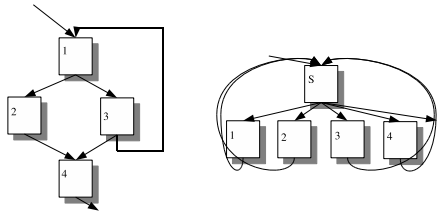

Figure 2. Basic control flow flattening

We implemented a form of control flow flattening in both TXL and SUIF. For this paper, we flattened nested if-then-else clauses, as illustrated in Figure 3. The main goal is to group the different conditions in a single value and to replace the nested if-then-else clauses by a single switch based upon this value. On the left hand side, two conditions have to be evaluated to know the control flow of the program. In the transformed control flow graph we force the attacker to analyze all three conditions. In particular, a new variable will be introduced and each bit of that variable contains the *true/false* state of each test condition of the if-then-else statement to be flattened. For the example in Figure 3, the test of the switch will be $(a\&\&1) << 2|(b\&\&1) << 1|(c\&\&1)$. Note that this means that both b and c will always be evaluated, regardless of the outcome of a. Therefore, we need to assure that b and c can be evaluated without side effects or errors. The last bit of the test condition will be condition c, while the second last one is the condition of b and the third last one is the state of condition a. Assuming that condition a is *true* and condition b is *true*, code block A will be executed in the original program. In the obfuscated program, the variable in the switch will evaluate to 6 or 7, depending on b. Both numbers will transfer control to code block A and the program is functionally equivalent to the original one.

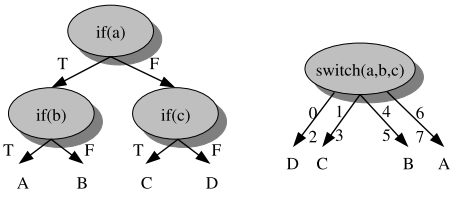

Figure 3. Flattening an if-then-else tree

Plenty of similar control flow flattening algorithms could be easily implemented into SUIF.

Irreducible Programs Most well-written programs are reducible which is an advantage for analysis, especially

identifying loops. Our obfuscator contains a transformation to make a reducible program non-reducible by duplicating a loop, and adding goto's to switch between both loops. This is illustrated in Figure 4. The transformation survives the compilation step, as long as $P^?$ is executable both ways. $P^?$ could be a random check because it is not important which direction will be taken, as both ways will execute the same code block *A*. Both paths could now be obfuscated differently and as a compiler can not optimize paths that are functional equivalent but syntactical different this obfuscation transformation will last in the final program.

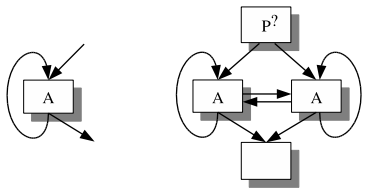

Figure 4. Making a program irreducible

4.3 Data Obfuscation

Understanding a simple algorithm such as sorting elements of an array is easy. Applying a simple data transformation on such algorithm can make it hard for someone to understand the code. We will apply a data transformations on the following piece of code:

```
for(i=0;i<10;i++)
  for(j=i;j<10;j++)
    if(a[j]>a[i])
      swap(a[i],a[j]);
```

Aggregation The first data transformation we would like to discuss is restructuring arrays. Arrays can be split, merged, folded or flattened [6]. We will merge two or more arrays into one:

Figure 5. Array transformations

Applying this transformation to our example will force the attacker to evaluate details of the algorithm if he wants to understand it. The test and swap lines will be transformed into the next piece of code, assuming that *a* is the array on the odd indices of the interleaved array.

```
if(a[2j+1]>a[2i+1])
  swap(a[2j+1],a[2i+1]);
```

Finding similar transformations for arrays is not hard and implementing them into the right tool neither. As it is already difficult in TXL to get type information, it makes this data transformation impossible to apply in a safe way. E.g., modifying a datastructure, requires the location of every instance of that data structure. On a parse tree this is non-trivial as the same name might be used in different scopes for different datastructures. While the parse tree does contain sufficient information to deduce the type of datastructure when, it is a more straightforward to perform this on an intermediate representation which contains a symbol table, such as SUIF.

Ordering An obfuscation transformation which reorders arrays is neither difficult in SUIF. A symbol table is at our disposal so each pointer to the array is known, which makes finding all accesses to the array straight forward. The indices used to access the array can be changed by a function mapping the original position *i* into its new position of the reordered array. The test and swap lines of our example will be changed into the next piece of code which will no longer order the array as in the original program. Although, all indices will be changed in the program, so the resulting code stays functionally equivalent with the original one.

```
if(a[f(i)]>a[f(j)])
  swap(a[f(i)],a[f(j)]);
```

Storage and encoding Data flow optimizations such as common subexpression elimination and constant propagation are able to undo very trivial data obfuscations. For example when splitting constant 10 into subexpression $2+8$, constant propagation will undo this transformation.

Non-trivial data obfuscations such as these shown above always survive the compilation process because these transformations change the context of the program. While a compiler only has optimizing transformations at his disposal, he is unable to undo such context changing data transformations. On the other hand is variable splitting a deoptimization transformation and applying such transformation should take into account the optimizations performed by the compiler.

We had a look at binary obfuscators [15, 12] and found out that no non-trivial data transformations were implemented. Only trivial data transformations such as constant splitting are implemented at binary level and without further obfuscation, an optimization run afterwards could remove these transformations. It is not astonishing that binary obfuscators only contain trivial data transformations as the types of datastructures are lost during compilation. Passing extra information to do such transformations at a binary level is feasible, but intensive and rather artificial if these transformations can be a source code level and afterwards survive the compiler optimizations. As the SUIF framework provides all the necessary information to properly apply data transformations, we encourage to perform data obfuscation at a source code level if the final goal is binary obfuscation.

4.4 TXL versus SUIF

The first advantage of TXL is the fact that the representation of the source code in a parse tree is close to the source

code itself. Also, the learning process of doing transformations on a parse tree benefits from the similarity between the original code and the internal representation of the code into TXL. But as this internal representation is strongly related to the source code, more complicated transformations are hard. Especially transformations that require type information are difficult to implement. Propagating information such as type information is necessary to safely apply obfuscation transformations but this approach resembles more the intermediate representation of SUIF. Identifying control flow in TXL has the same problems as trying to deduce type information. It is difficult to extract control flow information out of the parse tree and deriving it from the tree leads to an intermediate representation. The last disadvantage we discovered, is the fact that each transformation has to be applied on the entire file. It is only possible to prevent a transformation of going into an infinite recursive loop, but is hard to apply the transformation on some matches. Applying the transformation on some matches leads to variation and this could also confuse the attacker.

A medium-level intermediate representation such as the SUIF representation consists of type information available through a symbol table. This is a major advantage when applying data transformations. As SUIF is some sort of compiler, control flow can easily be followed through the program. A disadvantage is the fact that the intermediate representation is farther from the source code than the representation of the code in TXL.

5 Conclusion and Future Work

In this paper, we studied the effectiveness of source code transformations for binary obfuscation. We first gave a brief overview of the existing obfuscators and which code they obfuscate. Then, we implemented several obfuscation transformations into two source code transformation frameworks (TXL and SUIF) and we evaluated whether these transformations produced an obfuscated program after compilation. We illustrated that it is possible to apply source code obfuscation when binary code obfuscation is desired. We also concluded that SUIF is a powerful framework to implement data obfuscation transformations, while TXL is a simpler framework, straight forward to learn and suitable for basic, global obfuscation transformations.

Future Work As the SUIF framework provides all the necessary information to properly apply data transformations and data transformations are useful for binary code obfuscation, the future work is exploring all possibilities of data obfuscation at source code level to achieve binary code obfuscation. In literature, interesting data transformations are available, but only few are implemented in an obfuscation tool because data transformations require a extensive knowledge about the program and most of the obfuscators are unable to provide the necessary information to apply the obfuscation transformation.

Acknowledgments

The authors would like to thank the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) and the Fund for Scientific Research Flanders (FWO) for their financial support. This research is also partially supported by Ghent University, the HiPEAC network, and the by the Concerted Research Action (GOA) Mefisto 2000/06 of the Flemish Goverment.

References

[1] B. Anckaert, B. De Sutter, D. Chanet, and K. De Bosschere. Steganography for executables and code transformation signatures. In *Proc. 7th International Conference on Information Security and Cryptology*, pages 425–439, 2005.

[2] G. Arboit. A method for watermarking java programs via opaque predicates. In *Proc. of ICECR-5*, October 2002.

[3] I. Baxter, C. Pidgeon, and M. Medhlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *Proc. International Conference of Software Engineering*, May 2004.

[4] C. Collberg, G. Myles, and A. Huntwork. Sandmark - a tool for software protection research. In *IEEE Security and Privacy*, volume 1, July/August 2003.

[5] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages, POPL'98*.

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

[7] J. Cordy. Txl: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, January 1991.

[8] J. Cordy. Source transformations, analysis and generation of txl. In *Proc. of ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*, 2006.

[9] L. Ertaul and S. Venkatesh. Jhide-a tool kit for code obfuscation. In *Proc. of IASTED International Confrerence on Software Engineering and Applications (SEA 2004)*, 2004.

[10] S. C. Group. The SUIF Library: A set of core routines for manipulating SUIF data structures, Stanford University, 1999.

[11] S. V. Levent Ertaul. Novel obfuscation algorithms for software security. In *Proc. International Conference on Software Engineering Research and Practice (SERP05)*.

[12] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th. ACM Conference on Computer and Communications Security (CCS03)*.

[13] D. Low. Java control flow obfuscation. Master's thesis, University of Auckland, New Zealand, October 1998.

[14] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software protection through dynamic code mutation. In *The 6th International Workshop on Information Security Applications (WISA)*, 2005.

[15] M. Madou, L. Van Put, and K. De Bosschere. Loco: An interactive code (de)obfuscation tool. In *Proc. of ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*, 2006.

[16] A. Majumdar and C. Thomborson. Securing mobile agents control flow using opaque predicates. In *Knowledge-Based Intelligent Information and Engineering Systems: 9th International Conference, KES*, 2005.

[17] G. Myles and C. Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. In *ICECR-7*, 2004.

[18] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *16th Annual Computer Security Applications Conference (ACSAC00)*.

[19] C. Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, Department of CS, University of Virginia, 2000.

[20] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *International Conference of Dependable Systems and Networks*, Goteborg, Sweden, July 2001.

[21] K. S. Wilson and J. D. Sattler. Software control flow watermarking, Aug 2004. Baker and Botts, US2005/0055312 A1.

[22] G. Wroblewski. General method of program code obfuscation. In *Proc. International Conference on Software Engineering Research and Practice (SERP 02)*, pages 153–159, Las Vegas, USA.