Using Exception Handling to Build Opaque Predicates in Intermediate Code Obfuscation Techniques

Daniel Dolz Gerardo Parra

Grupo de Investigación en Robótica Inteligente
Departamento de Ciencias de la Computación
Universidad Nacional del Comahue
Buenos Aires 1400
(8300) Neuquén - Argentina

ABSTRACT

Microsoft's .NET Framework, and JAVA platforms, are based in a just-in-time compilation philosophy. Software developed using these technologies is executed in a hardware independent framework, which provides a full object-oriented environment, and in some cases allows the interaction of several components written in different programming languages. This flexibility is achieved by compiling into an intermediate code which is platform independent. Java is compiled into ByteCode, and Microsoft .NET programs are compiled into MSIL (Microsoft Intermediate Code). However, this flexibility comes with a price. With freeware tools available in Internet, it is quite easy to decompile intermediate codes and obtain a working, readable version of the source code. Obfuscation is the most accepted and commercially available technique that developers can use to protect their intellectual property In this work, we propose the use of try-catch mechanisms available in .NET as a way to improve the quality of one of the building blocks of obfuscation: opaque predicates.
Keywords: Obfuscation. Obfuscation Transformation. Opaque Predicates.

1. INTRODUCTION

JAVA applications, and applications developed under any version of Microsoft .NET Framework are vulnerable in the sense that it is possible for anyone with minimum knowledge to get a fully functional compilable version of the source code using downloadable free tools. Losing the source code in hands of an unauthorized organization with illegal intentions could have the following consequences:

- Your competence could acquire your money invested in I+D. Competence may, in an unloyal move, launch your very software with minor tweaks taking advange of your original investment.
- A competing organization could find flaws in the product and use them in their own benefit.
- In the special case of modern encrypting algorithms, whose strength relays in the existence of an unknown key instead of a particular set of instructions, getting access to source code could be useful to practice a brute force attack against them.
- Access to source code makes easier illegal pirate-style activities suck as key-cracking, expirations dates and harlocks tampering.
- A discontent employee may get the source code of any of the organization's application, modify the sql sentences in it, recompile it, execute it, and perform queries which she is not allowed to perform according to her level access. This could result in the loss of valuable commercial secrets, for example.

So far we have mentioned money issues, but some governments such as US government consider this vulnerability to be a national security issue[6]. In private industry, it is a well known fact that 75% of the Fortune 500[1] companies use Microsoft Visual Studio 2005[9]. Another well known fact is that the main menaces to organizations today are not external agents but inner personnel with access to company resources from the inside (employees, contractors, consultants, etc.). This is not a minor problem[11].
Our line of work, initiated in [4], shows how obfuscation techniques are the branch of Computer Security that provides the best protection level within the existing alternatives. In this article we show how, using run time exceptions, it is possible

to notoriously increase opaque predicates' quality, indeed increasing the quality of overall obfuscation and the protection of intellectual property.
This article is organized as follows. It starts with an introductory description of obfuscation and its key concepts. Then we describe the constructions called opaque predicates. In section 4, we present this work's proposal to create better opaque predicates. More advanced techniques are introduced where the concept of using exceptions using try-catch blocks instead of conditional branch sentences is presented. Finally, in section 5, we report the conclusions and lines for future work.

2. OBFUSCATION - KEY CONCEPTS

To obfuscate a source or an intermediate code is a process that performs a transformation using re-writing algorithms, from readable understandable code into a functionally-equivalent one but not readable or understandable for human readers. Figure 1 shows the process and concept of intermediate code obfuscation.

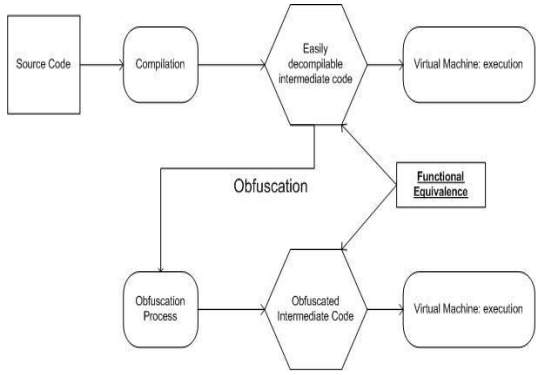


Figure 1: Intermediate Code Obfuscation.

In a nutshell, most frequent obfuscation techniques are the inclusion of irrelevant loops, unnecessary calculations, out of context checks, meaningless identifiers, useless functions, incredible relations, and so on. Other techniques are, nevertheless, way more powerful, in the sense that they require a deep knowledge of the platform being obfuscated. They can even be designed to mess with specific reverse engineering tools. Next, we discuss some key concepts introduced by Collberg[2].

Obfuscating Transformation
Let $P \xrightarrow{t} P'$ be a transformation of a source program P in a target program P'. $P \xrightarrow{t} P'$ is an obfuscation transformation if P and P' have the same observable behavior. More precisely, in order for $P \xrightarrow{t} P'$ to be a legal obfuscation transformation the following conditions must hold:

- IF P fails to terminate or terminates with an error condition, P' may or may not terminate.
- Otherwise, P' must terminate and produce the same output as P.

P' has certain features to difficult the understanding of the decompiled code. Observable behavior is defined loosely as "behavior as experienced by the user". This means that P' may have side effects (such as creating files, sending messages over the internet, etc.) that P does not, as long as this side effects are not experienced by the user.
Obfuscation techniques can be classified in four categories[2]:

- Lexical structure: identifier renaming, format tweaking.
- Data Obfuscation: embedded resources encryption, metadata encryption, string encryption, hierarchy modification, variable unification.
- Control Flow Obfuscation: control flow reconversion, sentence reordering, loop condition extensions.
- Preventive Obfuscation: meant exclusively to create malfunction or even crash known reverse engineering tools.

Quality of an Obfuscating Transformation
The quality of an obsfuscating transformation t is measured using four criteria: how hard is for human readers to understand the obfuscated code (potency), how hard is for an automated tool to revert the transformation (resiliency), how well the obfuscator's introduced code blends with the original code (stealth) and how much extra cost, if any, the obfuscation introduces (cost).

Potency: The potency of an obfuscation technique indicates, how harder is the obfuscated code to read for a human reader compared with the original source. The concept "harder to read" cannot be measured objectively, but nevertheless, software engineering metrics can be used. Such metrics measure conceptual clarity and maintainability of a referenced program source. For the purposes of this work, we will measure potency on a three point scale: high, mid and low.

Resilience: A serious attacker to intellectual property may have some reverse engineering tools, or may even develop or adapt her own, so resilience is expressed as the combination of two measures:

- **Programming Effort**: the amount of time required to construct an automatic deobfuscator that is able to effectively reduce the potency of a transformation t.
- **Deobfuscator Effort:** the execution time and space required by such an automatic deobfuscator to effectively reduce the potency of t.

It is important to distinguish between resiliency and potency. A transformation is potent if it manages to confuse a human reader, but it is resilient if it confuses an automatic deobfuscator.
Most resilient transformations are irreversible. Usually, they remove the information useful to humans but meaningless for a computer, such as identifiers. Other transformations, such as the addition of garbage code can be reverted with different levels of difficulty.

Stealth: It is possible to create potent and high resilient obfuscation techniques resulting in a very difficult to understand code. An example may be the modification of values in local variables. Instead of assigning the variable with the original value as the programmer intended, a valid obfuscation technique may assign enormous values in the range of millions and apply calculations in any location where the variable is used to maintain equivalence. With this technique, simple expressions such as $while(I \leq 10)$ may become $while((I * f(I) - 234)^{12} \leq 5748951478)$ being functionally equivalent. However, this new code is easily spotted as strange and alien, and a reverse engineering may identify it as obfuscator-introduced very easily. To improve stealth, obfuscator introduced code should look and feel like the non obfuscator introduced code in order to make an attacker believe it is valid code. Improving stealth is not an easy task, because a stealthy code in a domain may not be stealthy in another.

Cost: The application of many obfuscation techniques, i.e. the one showed above clearly implies an execution overhead due to the increase of sentences and calculations. Other transformations may increase the program requirements of memory compared with the original program. Some transformations do not have an overhead and they may even imply an optimization, such as identifier renaming. Higher cost means lower quality.

3. OPAQUE PREDICATES

This work focuses in opaque predicates. Opaque predicates are the basic blocks of obfuscating transformations that hide the control flow of the program[3]. Flow control transformations fall into three categories:

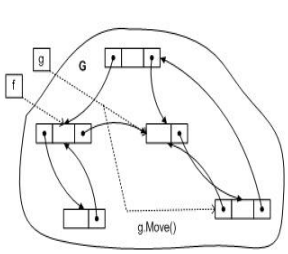
- Hide the real control-flow behind irrelevant statements that do not contribute to the actual computations.
- Introduce code sequences at the object code for which there exists no corresponding high-level language construct.
- Remove real control flow abstractions or introduce spurious ones.

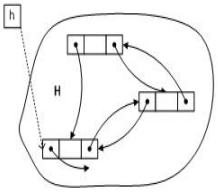
Opaque predicates are constructions that do not belong to the original source code but are introduced by the obfuscator. They are the real thing in the sense that they branch the run time execution into the real programmer code and not the obfuscator introduced spurious one.
Not formally, a variable V is opaque if V has a property which is known at obfuscation time but it is not known by a reverse engineer. The same stands for opaque predicates with boolean values known by the obfuscator but not by the reverse engineer. For example, an opaque variable V introduced by an obfuscator with value "10" may be used to generate true o false constructions by asking if $V == 10$, if $V < 6$, etc. The obfuscating tool knows the value of V at every point and it is under its complete control. However, the hardest for a reverse engineer to find out V's value, the better the obfuscation will work.
Opaque predicate creation is one of the biggest challenges obfuscator developers face. In fact, control flow transformations rely mostly on the quality of the opaque predicates. We'll use the same measures (potency, resilience, stealth and cost) to evaluate the quality of an opaque predicate.

Using Opaque Predicates

Opaque predicates in the following control flow transformations are critical:

- Dead or irrelevant code insertion. This code should never execute. Its existence is meant only to confuse a possible attacker. The non execution of the irrelevant code depends solely on an opaque predicate, i.e. dead code could be placed in an *else* block of an *if* sentence with an opaque predicate that always evaluate to true.
- Loop extension. It may be possible to obscure a source code by tweaking its termination condition. This can be done by introducing opaque predicates.






```
Node g,h;
method P(…, Node f)

/* 1 */  g= g.Move();
         h= h.Move();
/* 2 */  h= h.Insert(new Node);
         .
         .
         .
/* 3 */  x.R(…, f.Move());
         .
         .
         .
/* 4 */  if (f==g)?
/* 5 */  if (g==h)F
         .
         .
         .
/* 6 */  f.Token= False;
         g.Token= True;
         .
         .
         .
/* 8 */  f.Token= True;
         h.Token= False;
/* 9 */  if (f.Token)T
```

Figure 2: Opaque Predicates using pointers and alias.

A dynamic structure is built using nodes. Each node has a boolean token and to pointers that can reference other nodes. The structure starts with two connected components, G and H. There are two global g and h pointers that point to G and H respectively.

- Convert a reducible to a non reducible flow graph. Intermediate-code based platforms, such as the ones this works target (Java and .NET), are compiled to a virtual machine code which is more expressive than the language itself (Bytecode and IL, respectively). This is not coincidence, because it must be guaranteed that every higher level language construction can be converted to the intermediate code language. Language breaking transformations take advantage of this by introducing virtual machine instruction sequences which have no direct correspondence with any source language construct, but preserving correct execution using opaque predicates. An example could be a conditional branch to a sentence inside a while loop, protected by an opaque predicate that always evaluates to false in such a way that the correct execution is not tampered.

Manufacturing Opaque Constructs

The quality of most control flow transformations is directly dependent on the quality of opaque predicates. Obvious opaque predicate, such as $P = 0, Q \neq null$, etc. are not very resilient at all. This means that an automated deobfuscation tool could deduce its values using static analysis techniques without much effort. A higher level of protection is mandatory. Ideally, we would be able to create opaque predicates that require worst case exponential time (in the size of the program) to break but only polynomial time to construct.

Advanced Techniques

Advanced opaque predicate construction techniques exist. However, they are fundamentally flawed. One of the problems is the high overhead introduced, and the other is the visibility of the conditional jump sentence used to direct the execution. Next, we describe the most advanced techniques for building opaque predicates.

Opaque Constructs Using Objects and Aliases:

Interprocedural static analysis is significantly difficult whenever there is a possibility of aliasing. In fact, precise flow-sensitive alias analysis is undecidable in languages with dynamic allocation, loops and if-statements[5, 10]. Basic idea is the construction of a dynamic complex structure with alias and pointers keeping a set of pointers to that structure. In this way, the obfuscating tool knows if p and q are equal (being p and q pointers to a given structure) but an automated deobfuscating tool could not known it by the means of a static analysis. Figure 2 shows an example of a complex opaque construct structure.

Opaque Predicates Construction Using Threads: Parallel or multiprocess programs are way more difficult to analyze than sequential ones. The reason is their interleaving semantics: if n code regions can be executed in a parallel fashion, the program can be executed in $n!$ different ways. Depending on the size of n, this could be a huge number. The opaque predicate technique would be the same than with objects and aliases but adding the complexity of parallel execution.

Detected Problems

These are the most advanced techniques today. However, in this work we suggest that these techniques are not practical because they imply a very high cost and they are not stealthy at all.

Cost: In both techniques (objects and alias and multithreading) a graph structure is created which may be (and probably will be) totally alien to the original program just to the effect to ask, in some moment, if $P = Q$ or a question alike. Object-alias technique imply a big spatial cost (memory) and temporal cost (CPU cycles needed to create and maintain the structure). Multithreading is even worse, because there are context-switching overheads. There is a practical consideration also: software fails sometimes. When this happens, modern environments let the user know specific data such as stack traces, memory dumps, etc. Programmers may find this data

useful. However, an obfuscator-dirty stack trace will probably not be as useful as an uncontaminated one.

Visibility: In this work we suggest that it is not the complexity of the calculations of the opaque predicate that matters, but its visibility. It is actually possible to keep a huge, complex and expensive structure with the goal of keeping the reverse engineer from knowing if $P = Q$, but eventually, he will detect that P and Q are alien and are not related with the program being analyzed and he will reach to the conclusion that $P = Q$ is in fact an opaque predicate, even if he can't tell its value whenever it is used. Identifying the opaque predicates is already a very valuable asset when deobfuscating the program, because the reverse engineer knows that one of the two possibilities (true or false) may hide the real code, and this knowledge may be enough to accomplish her purposes. An initial tool to detect opaque predicates is nothing more than to detect every conditional branch sentence, knowing that some of them belong to the real program and some of them are obfuscator-created. This is trivial. In fact, `ildasm.exe` tool is already provided by Microsoft to extract IL intermediate code from .NET portable executable binary files. `Ildasm` sets a blank line after every conditional branch instruction, making really easy to identify potential opaque predicates.

4. SUPERIOR OPAQUE PREDICATES: USING TRY-CATCH-FINALLY BLOCKS

In this section, we introduce our proposal to create higher quality opaque predicates.

Try-Catch-Finally Blocks

Programming languages with need for obfuscation (Java, Visual Basic.NET, C#, managed C++) implement exception handling with *try-catch-finally* constructs. It is not our intention to describe this construction. Suffice to say that *try*, *catch* and *finally* are keyword that delimitate distinct code blocks in a way that, if any exception occurs inside a *try* block (exception may occur even in a different method in a different binary file) execution is automatically branched to the first sentence of the *catch* block. If no exception is raised, catch block is not executed. Finally block is optional. When present, it is always executed and is usually used to perform cleaning tasks in both cases (exception or not). Classic example of a *finally* block can be found in database connection closing.

Constructing Opaque Predicates Using Exceptions

Our proposal consists in using really simple and inexpensive opaque predicates (such as $q = 0$, $p = null$) but avoiding the usage of a conditional branch sentence and instead, forcing an exception. So, even if the opaque predicate works with simple values, it will be extremely difficult for a reverse engineer to identify the exact sentence where the branching happens using static analysis techniques. Next, we show examples using Microsoft's .NET Intermediate Language, named MSIL. Explaining MSIL is outside the scope of this work. ECMA specifications[7, 8] are available on the web.

Example 1: Opaque Predicate using an IF sentence. The construct is an *if* (Opaque Predicate FALSE) *then* (real code) *else* (bogus code)

```
IL_0000: ldarg.1
IL_0001: brtrue.s IL_003e /* conditional branch.*/
.......... /* Real app code */
IL_0033: ldstr "Real Code"
IL_0038: call  void
           [mscorlib]System.Console::Write(string)
IL_003d: ret
.......... /* Bogus Code */
IL_003e: ldstr "Bogus obfuscator-introduced code"
IL_0043: call  void
           [mscorlib]System.Console::Write(string)
IL_0048: ret
```

Notice line `IL_0001` (remarked). It performs the conditional branch instruction *brtrue* using the method argument 1 (`ldarg.1`). It should be noticed that even when it may be really difficult to find out the value of this argument, it is really easy to find out that the *brtrue* sentence in line `0001`. Next, analyzing both then and else blocks, a reverse engineer may deduce the real code.

Example 2: Opaque Predicate using *Try-Catch*

```
IL_0000:  ldc.i4.0
IL_0001:  stloc.0
try
{
  .......... /* Real code */
  IL_0032:  ldstr  "Real Code"
  IL_0037:  call void
          [mscorlib]System.Console::Write(string)
  IL_003c:  ldloc.0
  IL_003d:  ldarg.1
  IL_003e:  div
  .......... /* False Code */
  IL_003f:  call string
        [mscorlib]System.Convert::ToString(int32)
  IL_0044:  call void
          [mscorlib]System.Console::Write(string)
  IL_0049: ldstr "Bogus obfuscator-introd. code"
  IL_004e:  call void
          [mscorlib]System.Console::Write(string)
  IL_0043:  leave.s  IL_0032
}  // end .try
catch [mscorlib]System.Object
```

```
{ ..... /* Real Code */
  IL_0055:  pop
  IL_0056:  ldstr  " Real Code"
  IL_005b:  call void
          [mscorlib]System.Console::Write(string)
  IL_0050:  leave.s  IL_0032
}  // end handler
IL_0052:  ret
```

An interesting exercise may be to identify in which exact sentence the opaque predicate is, knowing that the first argument value is the same that in the previous example, an integer of value zero. Answer is, in line `IL_003e:` *div*. This sentence raises a *Division By Zero* exception. This results in the execution continuing in the first sentence of the catch block where the rest of the real code is. The important fact to notice is that the induced exception may be anywhere inside a code block, indeed not making use of easily spotable sentences such as conditional branch sentences. This makes the opaque predicate just another sentence, not identifiable by means of static analysis. Some of the candidate exceptions for opaque predicates are division by zero, invalid use of null, types mismatch, out of range conversions and invalid casts, between others.

Constructing Opaque Predicates Using Exceptions with Improved Stealth
Let's go farther with this idea. Given the fact that the execution bifurcation is achieved raising a controlled run time error, a good obfuscator may use commonly used constructions present in the program to build opaque predicates that look like this structures.

Example. Invalid Use of Null: Let's say the obfuscator detects that the program to obfuscate makes intensive use of objects that are instances of a class named *CACIC* and a method named *Share()*. This means that sentences like *A.Share()* are common in the program. *A* is a variable name and holds an instance of *CACIC*. The obfuscator could introduce opaque predicates in the form of sentences like *A.Share()*, but in a place where the value of *A* is known to be *null*. Control Flow obfuscation is correctly made: a reverse engineer will find only a code block that looks like the rest of the code, featuring frequently used constructs such as *A.Share()*. However, as *A* is null, execution goes to catch block. This is really difficult to detect with an automated static analysis, and maybe more difficult performing a visual analysis of the code, because the bifurcation occurs in a completely common sentence in the code. In conclusion, these are very stealthy opaque predicates.

Example in Cooperation With the Programmer: If the application to obfuscate access data located in a database, with a little help from the programmer, exceptions could be raised inducing erroneous SQL sentences. This would confuse an unsuspecting reverse engineer even more because it is not likely that an automated obfuscation had introduced SQL sentences in a program.

Limitations
This idea has some limitations. One of them is that *try-catch-finally* constructs can be nested one inside of another, but they can not be overlapped. This limits somehow the constructions that can be created. Another important issue is that original *try-catch-finally* must clearly be respected.

5. CONCLUSIONS

Obfuscation is the standard way to protect source code in modern development environments. Opaque Predicates are the building blocks of the Control Flow obfuscating transformations, and its quality determines the overall obfuscating quality. We have analyzed the most advanced techniques to build opaque predicates, such as alias and multithreading, and we concluded that, because of their cost, they are not the optimal solution. In the present work, we propose the creation of opaque predicates using exceptions handling in order to create high quality, inexpensive and stealthy opaque predicates. Future work: going deeper into the most advanced features of modern environments in order to find out new ways to protect intellectual property. There is also much to do in the concept of "*functional equivalence*" between the original program and the obfuscated counterpart.

References

[1] Fortune 500. http://en.wikipedia.org/wiki/fortune_500.

[2] Christian Collberg and Clark Thompson. Watermarking, tamper-proofing, and obfuscation - tools for software protection.

[3] Christian Collberg, Clark Thompson, and Douglas Low. Manufactoring cheap, resilient, and stealthy opaque construct.

[4] D. Dolz and G. Parra. Ofuscadores de código intermedio. Reporte preliminar. In *VIII Workshop de Investigadores en Ciencias de la Computación*, 2006.

[5] Susan Horwitz. Precise flow insensitive May-Alias analysis is NP-Hard. *TOPLAS*, 19(1):1–6, 1997.

[6] Jeff Hughes and Martin Stytz. Advancing software security - the software protection initiative, 2001.

[7] The Common Language Infrastructure (CLI) Partition II. Metadata Definition and Semantics.

[8] The Common Language Infrastructure (CLI) Partition III. Cil Instruction Set.

[9] Microsoft Software Developer Network. Microsoft Visual Studio 2005 Evaluation Guide.

[10] G. Ramalingam. The undecidability of aliasing. *TOPLAS*, 16(5):1467–1471, 1997.

[11] Revista Information Technology. Suplemento Especial Seguridad, 2005.