An Inter-Classes Obfuscation Method For Java Program

Xuesong Zhang, Fengling He, Wanli Zuo
College of Computer Science and Technology, Jilin University
Changchun, 130012, P.R.China
xs_zhang@126.com, Hefl@jlu.edu.cn, wanli@jlu.edu.cn

Abstract

Software is a valuable form of data, representing significant intellectual property, and reverse engineering of software code by competitors may reveal important technological secrets. This problem becomes more serious when facing with the platform independent language—Java byte code. We introduce an inter-classes software obfuscation technique which extracts the codes of some methods in user-defined classes and embeds them into some other object's methods in the object pool. Since all objects in the object pool are upcast to their common base type, which object's method will really execute can only be ascertained at runtime. Thus, drastically obscured the program flow. Combined with some enhanced mechanisms, this technique can even resist to dynamic analysis to a certain extent. Experimental result shows that there is little influence to the execution efficiency.

1. Introduction

In order to meet the platform independent characteristic, Java introduces a form of symbolic link technique, which stores all the type and interface information into class file's constant pool. These information provide the loading on demand and mobile property, but also facilitate decompilation at the same time. Faced with such a neutral Java byte code language, traditional software encryption technology gains little effect. In the premise of distributed computing, hardware-assisted encryption technology cannot provide code mobility, and will greatly limit the interoperability of Java application; Encryption through self-defined class loader cannot provide effective protection either, this is due to the implementation of Java programs. In order to carry out security checks, all of the class related byte codes must be completely loaded into memory before their execution, malicious users can directly extract the entire decrypted codes. At the same time, since byte code cannot access stack directly, and does not able to modify its own code dynamically, native code's self-encryption technology cannot be used for Java programs.

Under these circumstances, software obfuscation provides a more effective means of protection. Obfuscation can prevents end users from understanding a program's design and code, and therefore make reverse engineering uneconomical. The general idea of our proposed obfuscation scheme is as follows: one instance method invocation in Java language can be interpreted as a kind of special unconditional jump in assembly language level, and all those methods invocation can be transformed to a unified style, so long as they have the same parameter and return type. This form of transformation will lead to a strong obfuscation result by further using of alias and method polymorphism. We call this kind of obfuscation algorithm as inter-classes obfuscation. In Figure 1, the codes of some methods in user defined class are extracted and embedded into some object's methods in the object pool. All the objects in the class pool are inherited from the same super class, and their relations are either paternity or sibling. Each object's *DoIt* method is the mergence of more than two methods in user-defined classes. When the program going to execute one merged method which is originally defined in user-defined class, a request is sent to the class pool, and the class pool will return one object whose method is executed instead according to the request parameter. Since objects in the class pool are upcast to their common base type, which object's *DoIt* method will really execute can only be ascertained at runtime.

The rest of this paper is organized as follows: Section 2 reviews some related work. Section 3 describes the inter-classes obfuscation scheme in detail. Section 4 discussed some enhanced obfuscation mechanisms. In section 5, some experimental results are given. Finally, Section 6 concludes the paper with some future work.

978-0-7695-3126-7/08 $25.00 © 2008 IEEE
DOI 10.1109/ISA.2008.49

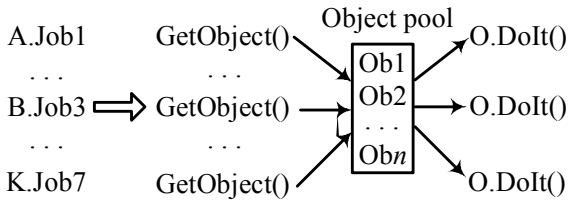

Figure 1 Object pool model

2. Related work

The first formal definition of obfuscation was given by Collberg et al. [1]. They defined an obfuscator in terms of a semantic-preserving transformation function *T* which maps a program *P* to a program *P*' such that if *P* fails to terminate or terminates with an error, then *P*' may or may not terminate. Otherwise, *P*' must terminate and produce the same output as *P*. Obfuscating transformation can classified into four categories:

Lexical obfuscation: Changes or removes useful information from the intermediate language code or source code, e.g. removing debugging information, comments, and scrambling/renaming identifiers. Lexical obfuscations are not alone sufficient because a determined attacker can infer the meaning of program identifiers from the context.

Layout obfuscation: Weakens the internal inherent logic between elements, e.g. changing the inherit relation of class, method in-lining, splitting and merging, etc

Data obfuscation: Targets data and data structures contained in the program, complicates their operations and obscures their usage, e.g. changing data encoding, variable and array splitting and merging.

Control-flow obfuscation: Alters the flow of control within the code, e.g. reordering statements, methods, loops and hiding the actual control flow behind irrelevant conditional statements.

Because pointer and goto statement are not support in Java, some obfuscating techniques which have a strong sense of protection[2, 3] cannot apply to Java program straightforwardly. Hence, obfuscating of Java program needs more refinement design. Currently proposed methods include:

Chan et al.[4] bring forward an advanced identifier scrambling algorithm. They utilize the hierarchy characteristic of jar package, i.e. a sub package or a top-level type (classes and interfaces) may have the same name as the enclosing package. Sequentially generated identifiers are used to replace those original identifiers in a package, and the generation of identifiers is restarted for every package. They also use the gap between a Java compiler and a Java virtual machine to construct source code level rules violation, such as illegal identifiers, nested type names. However, this kind of source code level rules violation can be repaired at byte code level by some automated tools[5]. Thus, only has limited protection ability.

Sonsonkin et al.[6] present high-level transformations of Java program structure—design obfuscation. Class coalescing replaces several classes with a single class. Class splitting replaces a single class with multiple classes. If class splitting is used in tandem with class coalescing, program structure would be changed very significantly, which can hide design concept and increase difficulty of understanding.

Sakabe et al.[7] concentrate on the object oriented nature of Java. Using polymorphism, they encapsulate all method parameters and return types through new defined classes to hide information, and introduce opaque predicates around new object instantiations to confuse the true type of the object. In fact, our proposed method is similar to this technique. However, their method is more focused on inner obfuscation of single object, with the expansion of the scope of protection, their approach will lead to a sharp increase in program size, and sharp decline in performance. In our scheme, more effort is spent on cross obfuscating among multiple objects, influence to program size and performance is relatively low.

3. Inter-classes obfuscation

In Java, an application consists of one or more packages, and it may also use some packages in the standard library or other proprietary libraries. The part of a program that will be obfuscated by the obfuscation techniques is called the obfuscation scope. In this paper, obfuscation scope only refers to those packages developed by the programmer himself.

3.1. Invocation format unification

There are usually different parameters and return types among different methods in a program. In order to perform inter-classes merging, their invocation formats should be unified. The two classes import in Figure 2 are used for this purpose. They encapsulate the parameters and return type for any method. In these two classes, all non-primitive data types are represented by some items in the object array *aO*. The *ParamObject* has a few more Boolean fields than the *ReturnObject*, they are used to select the execution branch after multiple methods have been merged. There are three flag in *ParamObject*, and thus at most four different methods can be merged into one *DoIt* method. The interface *DoJob* declares only one method *DoIt*, which uses *ParamObject* as its formal parameter

and *ReturnObject* as its return type. All the methods to be merged will eventually embedded into the *DoIt* method of some subclasses of *DoJob*.

```
public class ParamObject {
  public double[] aD;   public float[] aF;
  public long[] aL;     public int[] aI;
  public short[] aS;    public byte[] aY;
  public char[] aC;     public boolean[] aB;
  public Object[] aO;
  boolean flag1, flag2, flag3;
}
public class ReturnObject {
  public double[] aD;   public float[] aF;
  public long[] aL;     public int[] aI;
  public short[] aS;    public byte[] aY;
  public char[] aC;     public boolean[] aB;
  public Object[] aO;
}
public iterface DoJob {
   public ReturnObject DoIt (ParamObject p);
}
```

Figure 2 Unification of invocation format

```
public class A {
  public int DoJobA1(int x){ . . . }
  public long DoJobA2(double x, double y){ . . . }
}
public class B {
  public int DoJobB1(int x, int y){ . . . }
  public char DoJobB2(String s){ . . . }
  public boolean DoJobB3(boolean b, char c){. . . }
}
{
   a = new A();     b = new B();
   a.DoJobA1(10);     b.DoJobB3(false, 'A');
}
```

Figure 3 Original classes definition and method invocation

```
public class DoJob1 implements DoJob {
  public ReturnObject DoJob(ParamObject p) {
    ReturnObject o = new ReturnObject();
    if(p.flag1){   //DoJobB2    }
    else if(p.flag2){    //DoJobA1    }
    else{    //Garbage code    }
    return o;
  }
}
```

Figure 4 Inter-classes method merging

3.2. Inter-classes merging

In determining which method can be merged, some factors such as inheritance relation and method dependency relation must take into consideration. Methods in the following scope should not be obfuscated.

- Method inherited from (implements of an abstract method or overrides an inherited method) super class or super interface that is outside of the obfuscation scope.
- Constructor, callback function, native method and finalize method
- Method declaration with throws statement
- Method which access inner-class
- Method inside which invoke other non-public method which inherited from super class or super interface that is outside of the obfuscation scope.

Two classes are defined in Figure 3, *A* and *B*. Figure 4 shows a possible merging instance. The method *DoJobA1* of class *A* and the method *DoJobB2* of class *B* are merged into one *DoIt* method. Those flag fields not used in the *ParamObject* can be used to control the execution of garbage code, which forms a kind of obfuscating enhancement. The garbage code here refers to the code that can executes normally, but will no destroy the data or control flow of the program.

Method polymorphism: If the merged method is inherited from super class or super interface that is inside of the obfuscation scope, all methods with the same signature (method name, formal parameter type and numbers) in the inherited chain should also be extracted and embedded into some *DoIt* methods respectively.

Method dependency: The merged method invokes other method defined in current class or its super class, such as there is an invocation to *DoJobA2* inside method *DoJobA1*. There are two approaches:

- If *DoJobA2* is a user-defined method, it can be merged further, or else modify its access property to public, and do the same as the second approach.
- Put the instance a of class *A* into the object array of *ParamObject* as a parameter, and use explicit binding inside the corresponding *DoIt* method, i.e. *DoJobA2* is converted to *a*. *DoJobA2*.

Field dependency: The merged method uses the field defined in current class or its super class. There are two approaches also:

- Add determiners for fields accessed by this method, which is similar to the second approach in method dependency. But this approach is not suitable for the non-public field which inherited from super class that is outside of the obfuscation scope.
- Add *GetFields* and *SetFields* method for each class. The *GetFields* returns an instance of *ReturnObject*

which includes fields used by all methods that are to be merged, and this instance is put into the object array of the *ParamObject*. Code in *DoIt* method can use this parameter to refer to the fields in the original class. After the execution of *DoIt*, an instance of *ReturnObject* is transferred back by invoking the SetFields method which making changes to the fields in the original class.

3.3. Object pool

A lot of collection data types provided by JDK can be used to construct the object pool, such as List, Set and Map etc.. However, these existing data types have standard operation mode, which will divulge some inner logical information of the program. We use the universal hashing to construct the object pool here.

The main idea behind universal hashing[8] is to select the hash function at random from a carefully designed class of functions at the beginning of execution. Randomization guarantees that no single input will always evoke worst-case behavior. Because of the randomization, the algorithm can behave differently on each execution, even for the same input, guaranteeing good average-case performance for any input.

Definition 1: Let H be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m - 1\}$. Such a collection is said to be universal if for each pair of distinct keys $k, l \in U$, the number of hash functions $h \in H$ for which $h(k) = h(l)$ is at most $|H| / m$.

One construction algorithm for a universal class of hash functions is: Choosing a prime number p large enough so that every possible key k is in the range 0 to $p - 1$, inclusive. p is greater than the number of slots in the hash table m, Let Z_{p1} denote the set $\{0, 1, ..., p - 1\}$, and let Z_{p2} denote the set $\{1, 2, ..., p - 1\}$. For any $a_1 \in Z_{p1}$ and $a_2 \in Z_{p2}$, the following equation makes up of a collection of universal hashing functions.

$$h_{a_1,a_2}(k) = ((a_1 k + a_2) \bmod p) \bmod m$$

We use universal hashing table to store all instances of classes inherited from *DoJob*. If collision occurs, second level hashing table is established in the corresponding slot. The randomizahion characteristic of universal hashing enable us assign different values to a_1 and a_2 each time the program start. Some expressions are constructed based on a_1 and a_2, and used as the key to access hashing table. Under this condition, the key is no longer a constant, and better information hiding result obtained. Figure 5 shows the structure of hashing table. Instance of class *DoJob9* is stored by key *keym*, and the same instance is acquired by key *keyl*. Notice that the key used to store an object is different from the key to request the same object, their relation and hashing table itself may be protected by other data or control obfuscating algorithm. In Figure 6, invocation to class *A*'s mehod *DoJobA1* is replaced by invocation to *DoIt* method in one of *DoJob*'s subclass.

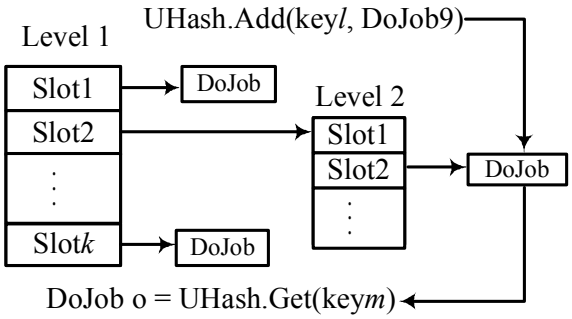

Figure 5 Structure of hashing table

```
DoJob1 dojob = new DoJob1();
UHash.Add( 217, dojob);
. . .
//a.DoJobA1(10)
ParamObject p = new ParamObject();
int[] i = new int[1];    i[0] = 10;
p.aI = d;    p.flag2 = true;
DoJob dojob = UHash.Get( 3 * a1 + a2 + 13 );
ReturnObject r = do.DoIt(p);
```

Figure 6 Invocation to class A's mehod DoJobA1 is replaced by invocation to DoIt method in one of DoJob's subclass

However, using key to access hashing table directly will cause some problem when in face of method polymorphism. Consider the inherited relation in Figure 7, if all methods in class *A* are extracted and embedded into some *DoIt* methods, method extraction should be performed during ecch method overridden in subclasses of *A*. Due to the complexity of points-to analysis, it's hard to determine which instance *a* will refers to in the statement *a.DoJob1*. As a result, it cannot be determined either that which key should used to access the hashing table. Under this situation, one method *GetIDs* should append to the super class A. *GetIDs* will return an array includes all keys corresponding to those methods in current class which have been merged into the object pool. If subclass overrides any method of parent class, the *GetIDs* method should also be overridden. Figure 8 shows the return arrays of each class corresponding to Figure 7. All IDs of the overridden method have the same position in the array as the original method in super class. In this way, the statement *a.DoJob1* can be replaced by invocation to *Uhash.Get* with the first element in the array as the parameter.

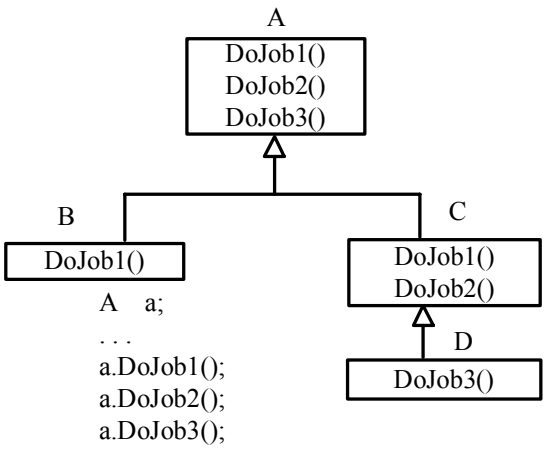

Figure 7 Method polymorphism

A	ID1	ID2	ID3
B	ID4	ID2	ID3
C	ID5	ID6	ID3
D	ID5	ID6	ID7

Figure 8 Array returned from different class by calling its GetIDs method

4. Obfuscating enhancement

In order to perform effective attack, which instance each *DoJob* references to should be precisely located. Since all objects added in to the object pool have been upcast to their super class *DoJob*, and different keys are used to store and access the same object in hashing table, relying solely on static analysis it is not feasible. However, frequently accessing of hashing table, and the if-else block partitioned according to flag in *DoIt* method still leak some useful information to a malicious end user. These information may be used as the start point of dynamic attack. There are many mechanisms to hide these information.

Multi-duplication: Make multi duplication to some merged methods. Each duplication is transformed by different algorithm to have distinct appearance, such as parameter type conversion, variable or array compressing and decompressing, splitting and merging. Every time a merged method execute, select different object has the same functionality.

Method interleaving: Methods merged into the same *DoIt* are branch selected by the flag value, bears the obvious block characteristic. Further action may be taken to interleave these blocks, obscuring their boundaries. The Boolean type flags can be obscured at the same time, e.g. importing opaque predicate, replacing one Boolean type with two or more integer types.

Random assignment of parameters: Since only parts of the fields in *ParamObject* are used during one execution of a merged method, they may be used to perform pattern matching attack by malicious users. Those unused fields can be randomly assigned any value before invocating to *DoIt*, and some garbage codes which use these fields are added.

Hashing table extension: Extend some slots to insert new objects. The parameters used in *DoIt* method of these newly inserted object are different from those used in the already exists object's *DoIt* method. When locating the slot which has more than one object by the given key, randomly select and return one. Before enter the corresponding execution branch according to the given flag, a check will be made to ensure whether those formal parameters in *ParamObject* are valid or not, including fields used by following instructions should not be null, and fields not used should be null. If parameter mismatch found, an exception is thrown. Now the *DoIt* invocation code is enclosed in a loop block (Figure 9), and following instruction will not execute until a success invocation.

```
while(true){
    try{
        dojob = UHash.Get( 572 );
        r = do.DoIt(p);
        break;
    }catch(ParamMismatchException e){
        continue;
    }
}
```

Figure 9 The DoIt method invocation model after hashing table extension

Dynamic adjusting of object pool: Multi-rules can be adapted to construct the object pool. Randomly select one operation rule at start point, and introduce a new thread by which readjust the operation rule once in a while. The key used to access object pool should also be modified along with the rule change. Clearly, combined with the previous mechanism, this enhancing measure can withstand dynamic analysis to a certain extent.

5. Experimental result

We extend the refactor plugin in Eclipse, and apply our scheme to five java programs. Currently, only the basic obfuscating method is implemented, not including those enhanced mechanisms such as multi-duplication, method interleaving etc. Some of the programs are Java applets which will never terminate without user action. We only measure their execution

Table 1 Experimental result by using only the basic obfuscation method

Program	Description	Method Merged		Before obfuscation	After obfuscation	Ratio
WizCrypt	File encryption tool	8	Jar file size (byte)	13755	21892	1.58
			Execution time (sec)	50.46	51.25	1.02
MultiSort	Collection of fifteen sortin algorithms	17	Jar file size (byte)	14558	23497	1.62
			Execution time (sec)	102.06	107.83	1.06
Draw	Draw random graphs	11	Jar file size (byte)	16772	26123	1.56
			Execution time (sec)	6.12	6.23	1.02
ASM	Artificial stock market	29	Jar file size (byte)	87796	97149	1.11
			Execution time (sec)	31.20	32.38	1.04
DataReport	Report generator for electric power data	22	Jar file size (byte)	59030	68555	1.17
			Execution time (sec)	8.71	9.15	1.05

time in finite cycles. For an example, the ASM program will simulate the stock market forever, and the corresponding execution time given in Table 1 is based on the first thirty cycles.

The experimental environment is as follows.

- Processor: Intel Pentium IV, 2.4GHz
- Memory: 1G RAM
- OS: Windows 2003 Service Pack 1
- Compiler: j2sdk-1_6_0_10

From Table 1, the program size increasing ratio after obfuscated lies in between 1.11 and 1.62. With the size growing of the original program, the ratio presents a downward trend. This is because of the fact that all newly inserted codes are mainly used for object pool definition and operation, while the codes used for method merging and invocations are relatively few. The largest execution time decline is no more than 6%. In fact, some of the merged methods are invoked more than 10000 times, such as the *join* method in MultiSort. However, since all objects in the object pool have been initialized soon after program start, accessing to the object pool the will only return an object which has already been instantiated. And at the same time, the classes *ParamObject* and *ReturnObject* are directly inherited from *Object*, apart from the need for loading, linking and initialization during their first creation, the follow-up instantiation is only a small amount of work. Thus, our proposed scheme has little performance influence to the original program.

6. Conclusion

In this paper, we introduced a software obfuscation scheme by inter-classes method merging, which is applicable to any object oriented language. In the next, we will implement those enhanced mechanisms discussed in Section 4, and apply a various forms of attacks to evaluate their actual security. Source code level obfuscation can be regard as a special kind of software refactoring, how to apply more refactoring techniques to software obfuscation also need further research.

References

[1] Collberg, C., Thomborson, C., and Low, D., A Taxonomy of Obfuscating Transformations. Technical Report#148. 36 pp. Department of Computer Science, The University of Auckland, New Zealand. 1997.

[2] Wang, C., Hill, J., Knight, J.C., and Davidson, J.W., Protection of software-based survivability mechanisms. *In Proceedings of the 2001 conference on Dependable Systems and Networks. IEEE Computer Society*. Pages 193-202. 2001.

[3] S. Chow et al., An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs, *Proc. 4th Int'l Conf. Information Security,* LNCS 2200, Springer-Verlag, pp. 144-155, 2001.

[4] Chan, J.T. and Yang, W., Advanced obfuscation techniques for Java bytecode, *Journal of Systems add Software* 71, No.2. pp. 1~11, 2004.

[5] Cimato, S., De Santis, A., and Ferraro Petrillo, U., Overcoming the obfuscation of Java program by identifier renaming, *Journal of Systems and Software* 78, pp. 60-72, 2005.

[6] M. Sosonkin, G. Naumovich, and N.Memon, Obfuscation of design intent in object-oriented applications. In *DRM '03: Proceedings of the 3rd ACM workshop on Digital rights management*, pages 142–153, New York, NY, USA, 2003. ACM Press.

[7] Y. Sakabe, M. Soshi, and A. Miyaji, Java obfuscation with a theoretical basis for building secure mobile agents. In *Communications and Multimedia Security*, pages 89–103, 2003.

[8] J. Lawrence Carter and Mark N. Wegman, Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2): pp. 143–154, 1979