Unifying the Method Descriptor in Java Obfuscation

Weiwei Liu
State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications,
Beijing, China
e-mail: artxiaowei@163.com

Wenmin Li
State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications,
Beijing, China
e-mail: liwenmin02@outlook.com

***Abstract*—Although there have been a lot of obfuscation technologies, the bytecode still remains a certain amount of information available for the use of reverse engineering. For this, we propose a new obfuscation method focusing on the MethodDescriptor representing the parameters that the method takes and the value that it returns, and discuss the suitable solution in some special conditions. Our evaluation results show that the new method can obfuscate the MethodDescriptor effectively, and the overhead is controlled in an acceptable range. In the field of software engineering, the obfuscated procedures have more excellent performances on various metrics such as Cyclomatic Complexity.**

Keywords-obfuscation; unify; method descriptor; bytecode

I. INTRODUCTION

For maintaining the platform independence, Java program is compiled to the bytecode file. However, as the intermediate code, it retains a large number of program information, and the attacker can easily rebuild the program using the reverse engineering.

Many obfuscation techniques have been proposed to harden reverse engineering attacks. Some over use an identifier [1], some propose several techniques to introduce syntactic and semantic errors into the decompiled program [1-2], some propose methods to change control-flow [2-3], and some reform the relationship between classes [4].Combination of the above, the bytecode still remains a certain amount of information available for the use of reverse engineering. Here, we focus on the *MethodDescriptor* [5].

Supposing there is an attacker who want to find a key method in the numerous codes. If the attacker has known serial possible *MethodDescriptor* of that the method, he can quickly narrow the search scope, achieve the fast locating.

For this problem, we take design one step further. In order to smooth out differences among method descriptors, we propose Unifying Method Descriptor (UMD) to make *MethodDescriptor* without differentiation. UMD strives to modify parameter types and return value type to a uniform format, and to correct the related code which calls the transformed method. And then, we develop an obfuscation program using the UMD algorithm and conduct a complete evaluation.

II. PROPOSE METHOD

In this section, we propose a new obfuscation algorithm UMD to against reverse engineering. Our ultimate purpose is to smooth out differences among methods' descriptor shown as Fig. 1[1].

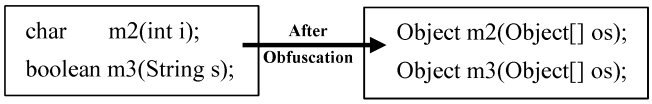


Figure 1. The effect of UMD

Due to the significant difference on method descriptor between m2 and m3 on the left, if we know that the target method's parameter type is *String* and the return value type is *boolean*, then, we can easily locate the method m3. When the procedure is obfuscated by UMD, as is shown on the right, there isn't any different on method descriptor between m2 and m3. Their parameter types have been modified to *[O*, and return value types have been modified to *O*. In this situation, even if we know the target method's descriptor, we cannot achieve the fasting locating.

For completing the design above, we will elaborate the following aspects.

A. Modify Method Definition

In order to unify the *MethodDescriptor*, modifying the method definition is the first and most important step. Firstly, we take a simple method as example.

```
int m1(int i){          0  iinc 1 by 1
        i++;            3  iload_1
        return i;       4  ireturn
}
```

Left part shows the source code of a method, and the corresponding bytecode is shown in the right part.

For remaining the method logic unchanged, we just add or modify some instructions in the position of **entry** (address 0) and **return** (address 4).

Firstly, supposing the parameter types is changed to *[O*, we need extract the original parameters in sequence from the *[O* before the **entry**.

[1] In order to facilitate the expression, ***[O*** denotes ***[Ljava/lang/Object;*** and ***O*** denotes ***Ljava/lang/Object;***, and these rules apply to all the *paper*.

Secondly, we need add type casting instruction before every **return** instruction to turn the return value type into *O*.

Finally, we modify the *MethodDescriptor* in the class file, and change it from *(I)I* to *([O)O*.

```
// extract the original parameters from [O
iinc 1 by 1;
iload_1
// turn the return type into O
```

B. Modify Method Invocation

After modifying all method definitions as above, we need traverse the entire procedure to correct the related code which calls the transformed method. There are two steps when modify each call instruction: **merging parameters** and **restoring return value type**.

Merging parameters

Supposing the called method is named *m2*, the *MethodDescriptor* is *(T1)T2*. If the *MethodDescriptor* has been modified to *([O)O*, here we should do something to merge the parameter *T1* into *[O*, and then put the *[O* into the operand stack.

In the virtual machine instruction, the pseudo instruction of method invocation is like following.

```
load T1                 //Push value of T1
invokevirtual #39       // m2(T1T2)T3
```

According to the bytecode, when JVM execute the *invokevirtual* instruction, the elements on the operand stack from top to bottom are [*T1*, *(other elements)*].

If we intent to push the parameter *T1* into the *[O*, in the normal instructions, bytecode is shown below.

```
iconst_1
anewarry #3
putstatic #18           //new Object[1];
getstatic #18
iconst_0
load T1
invokestatic #20
aastore                 //o[0] = T1;
```

According to current *MethodDescriptor*, we should combine the above two pieces of bytecode into following form.

```
load [O                 //Push value of [O
invokevirtual #39       // m([O)O
```

To simplify the complexity of modification, we insert the new instructions into just one position, where is before *invokevirtual #39,* to achieve the new form. Given the *T1* been already existed on the top of the stack, we have to use the *swap* instruction to swap the top elements to complete the storage operation for parameters. Specially, when the top element is of type *long* or *double*, because of two units it occupied and only one unit that the *swap* instruction can operate, we should use the *dup_x2* and *pop* to implement the function of *swap*.

Restoring return value type

As mentioned above, the return value type of *m2* has been changed to *O*. To ensure that the return value can be received correctly, after the call instruction, we have to use the *checkcast* instruction to restore the return value type from *O* to *T2*.

The two steps of **Modify Method Definition** and **Modify Method Invocation** above are the core ideas of UMD. However, only these are not enough. There are a lot of polymorphic concepts existed in Java program, such as *overloads*, *overrides*, and *inheritance*. To work with such features, we will continue to discuss flowing.

C. Overload Method

The method overload, is that you can create multiple methods with the same name in a class, but with different parameters and definitions, the return value type can be the same or different.

For two or more overload methods in one class, we cannot simply and directly modify their definitions, that would result in multiple methods with the same name and descriptor appeared at one class.

The solution used in this paper is to merge these overloads.Firstly, to adjust the structure of parameter *[O*, and vacate the position of index 0, where is used to store a digital ID to distinguish methods. Method parameters are stored backwards starting from index 1. Secondly, to merge the code segments in these overload methods using *ifelse* statement in serials, with the digital ID being the judgment condition.

D. Inheritance

Because of the inheritance, the subclass can inherit some methods from the superclasses. Thus, for a set of overload methods, the definition of several methods may exist in the subclass, while the other methods may exist in the superclasses. According to the UMD, whatever the part of subclass or superclasses, all these overload methods will be merged into a new method respectively. And then, the two new methods existed in subclass and superclasses will be a new relationship called *override*. And then, the problem comes.

```
class C1{
  byte a(
     int i){
   return i;
  }
}
```

```
class C1{
  Object a(Object[] o){
   int judge = (int)o[0];
    if(judge==0){
      int i = (int)o[1];
      return Integer.valueOf(i);
    }else{
      return null;
    }
  }
}
```

```
class C2 extends
C1{
  byte a(
      byte b){
   return b;
  }
}
```

```
class C2 extends C1{
  Object a(Object[] o){
   int judge = (int)o[0];
   if(judge==0){
      byte b = (byte)o[1];
      return Byte.valueOf(b);
    }else{
      return null;   // return super.a(o)
    }
  }
}
```

As is shown in the left segment, supposing there is a set of overload methods *a(I)I*, *a(B)B*, the *a(I)I* existed in superclass *C1*, the *a(B)B* existed in subclass *C2*. When we call the method *a(I)I* using the reference of subclass *C2*, JVM will execute according following steps:

Match a(I)I in C2 → *Match a(I)I in C1* → *[success]*

However, after obfuscation, like the right segment, *a(I)I* is merged to *a([O)O*, and *a(B)B* is also merged to *a([O)O*. As a result, the method *a([O)O* in *C2* override the *a([O)O* in *C1*. We perform a call same as above at this time, the JVM will goes wrong:

Match a([O)O in C2 → *[success]* → *Match digital ID of a(I)I in a([O)O* → *[failure]*

The mistake is caused by the method *a([O)O* in superclass *C1* been overridden by the same method in subclass *C2*, that is, the original method *a(I)I* existed in superclass *C1* disappeared.

For this problem, we propose such solution: When merging methods, we detect whether it has other overload methods in superclasses. And if so, we insert a statement *super.xxx()* in the last *else* block to call the overridden method in the superclass, for example, we replace ***return null*** with ***return super.a(o)***.

III. Evaluation

A. Limitations and Restrictions

Considering the correctness of obfuscated procedure, it should be noted that not all methods can be obfuscated in our obfuscation program. The following constraint conditions are given:

1. If a method *m1* is the implementation of an abstract method *m2*, the m1 cannot to be obfuscated when the m2 cannot too.
2. Some external methods and interfaces such as *Main(String[] args)* is not allowed to be obfuscated.
3. The instance initialization method <*init*> and the initialization method of a class or interface <*clinit*> is not allowed to be obfuscated.

With these restrictions, we implemented the UMD with Java language.

B. Benchmarks

For finding out the difference between original procedure and obfuscated procedure, we introduce the TamiFlex [6]. TamiFlex is a tool suite to facilitate static analyses of Java programs that use reflection and custom class loaders. The suite consists of two agents, one Play-out Agent and one Play-in Agent.

With the Play-out Agent you can monitor a Java program and dump all classes that the virtual machine loaded on this run to a specified directory.

With the Play-in Agent you can cause the virtual machine to load classes from specified directory instead of from they would normally be loaded from.

In short, the two agents' function is to dump classes and load classes.

We used the DaCapo 9.12 benchmark suite [7] to evaluate the protection effectiveness and the performance efficiency of our obfuscations. This suite consists of 14 real-world applications.

However, among the 14 procedures, the class loaders of *tradebeans* and *tradesoap* cannot only load classes from a fixed classes set, that lead to a changing input classes set for our obfuscation program. The implementations of *tomcat* and *eclipse* all involve some modifications in the class loader architecture, and the code is closely linked to the core library. We, therefore, cannot proving correctness of the obfuscations' result. In addition, the original procedure of *xalan* doesn't work with the Play-in Agent.

Get rid of the above 5 procedures which do not meet the requirements, we select the remaining 9 as test cases. Firstly, we dump all classes of each procedure respectively with Play-out Agent as the sets to be obfuscated. Secondly, using the obfuscation program to obfuscate each class set. Finally, we load the obfuscated classes instead of the original classes and compare the obfuscated procedures with the original procedures in Table I.

TABLE I. OVERVIEW OF DACAPO 9.12 BENCHMARKS PRE AND POST OBFUSCATION

Benchmar	execution time (MS)		code size (KB)		Obfuscated methods
	pre UMD	post UMD	pre UMD	post UMD	
sunflow	1093	2847	217	204	828/1022
pmd	604	645	383	393	2249/2807
h2	1469	3332	621	696	3613/4280
batik	2185	2868	1238	1231	4788/6684
luindex	445	705	312	309	1408/1740
lusearch	1192	2454	273	278	1352/1665
fop	1751	2901	1294	1303	4455/5665
jython	524	640	2450	2004	339/494
avrora	1901	28044	515	561	1973/2566

From Table I, the methods be successfully obfuscated account for about 70 to 80% overall. This shows that the 9 procedures were transformed widely and the obfuscated procedures can be used for experimental analysis.

Furthermore, after analysis, we find the most remaining methods not be obfuscated are restricted by condition 2 and condition 3, just a few are limited by condition 1.

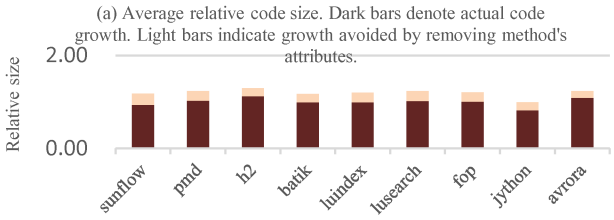
C. Overhead

Fig. 2(a) shows the relative size of the obfuscated procedures, in which the dark bars show the actual size changes with obfuscation program, the light bars on top indicate the code size saved by means of removing method's attribute. We can see that if just applying UMD obfuscation on procedures, the size will rise about 20-25%. If appending the means of removing attributes, the size will not change or even declined.

Fig. 2(a) shows the relative size of the obfuscated procedures, in which the dark bars show the actual size changes with obfuscation program, the light bars on top

indicate the code size saved by means of removing method's attribute. We can see that if just applying UMD obfuscation on procedures, the size will rise about 20-25%. If appending the means of removing attributes, the size will not change or even declined.

Fig. 2(b) shows the relative time of the obfuscated procedures. For most benchmarks, the performance overhead is very limited. For *avrora*, however, the median slowdown is 930 percent. For the UMD obfuscation, the additional time focusing on compounding and disassembling the *[O* when generating a method invocation. Thus, if there are intensive method invocations in somewhere, or there is a loop, in which it continuously calls a same method, these features are very likely to result to a skyrocketing on time. To prevent this extreme situation and control the runtime within certain limits, we can add some high-frequency and worthless methods to the prohibit list before starting the obfuscation program.



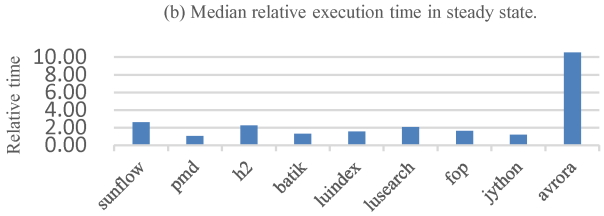


Figure 2. The overhead of 9 procedures after obfuscation

D. Provided Protection

Like all obfuscation researchers, we face the problem of measuring our techniques' potency. And as in all of the literature except a few studies involving human subjects, we know of no direct metrics to measure the resistance to reverse-engineering. So instead we rely on established software complexity metrics from the domain of software engineering. *CKJM* [8] is a tool for calculating Chidamber and Kemerer Java Metrics [9], and the *CKJM extended* is an extended version. The program *CKJM extended* calculates nineteen size and structure software metrics (include the CK metrics, QMOOD metrics [10] and others) by processing the bytecode of compiled Java files. Here, we use the *CKJM extended* to analysis the nine procedures before and after obfuscation. The change of the nineteen metrics is shown in Fig. 3.

We analyze the changes on each metric. Some important results is shown in Table II.

TABLE II. ANALYSIS RESULTS

WMC	This metric declined in all nine procedures. The UMD will merge all overload methods which in the same class, that means a non-increasing number of methods at least. The more number of overload methods, the more decreased degree of this metric. The maximum decline, about 20%, indicate that there exist quite a bit of overload methods in *pmd*.
CBO **Ca** **Ce** **IC** **CBM**	The five metrics all related to the coupling. The coupling includes stamp coupling, data coupling, control coupling and the like. The main work of the UMD is to combine several parameters to an object array. During this process, the data coupling and stamp coupling declined. Meanwhile, the merging of overload methods will weaken the control coupling between classes or between methods. Above all, the five metrics declined.
RFC	The metric measures the number of different methods that can be executed when a method is invoked for that object. It rose by 10%, but why does it rise after we merged a certain number of methods? In fact, we should not only consider the reduced methods, but also want to consider another aspect, that is during the process of compounding and disassembling *[O*, we have to invoke many type conversion methods, such as *Integer.valueof()*, these calls will significantly increase the RFC.
LCOM	This metric is calculate by: (n*(n-1)/2-k), n represent the number of methods, k represent the number of method pairs that share at least one field. It follows that there is a direct relationship between LCOM and WMC, and the LCOM presents the overall downward trend too. However, the declined degree of *luindex*, *lusearch* and *avrora* is larger than *pmd*. According to analyze, we find there are less overload methods or method pairs sharing at least one field in these three procedure, and these factors lead to a little change on k while the n declined sharply, and then the LCOM will declined sharply too.
LOC	From this metric, we know the number of code lines increased by 80 to 200%. Particularly, the major of increased code is used to compound and disassemble the *[O*.
CAM	This metric computes the relatedness among methods of a class based upon the parameter list of the methods. The value close to 1.0 is preferred. (Range 0 to 1). Owing to the main work of unifying the method descriptor, all obfuscated methods should have the same parameter list that is *[O*. Thus, this metric is up 40%.
AMC **CC**	The two metrics indicate the complexity of procedures. During the obfuscation process, we merge several different methods into one method. At the same time, for keeping original logic, we also add many *ifelse* statements and type conversion methods in the final merged method. These operations all lead to a more complex procedure.

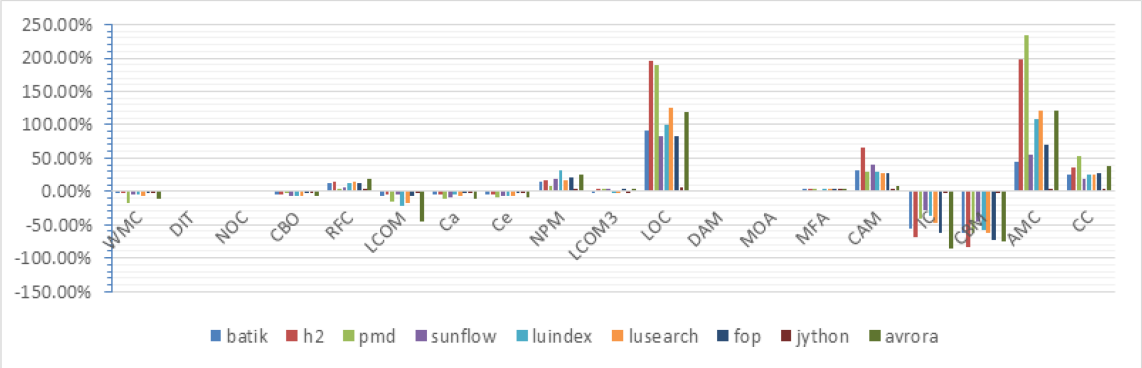


Figure 3. Reflect changes of 19 metrics on the 9 procedures after obfuscation.

After analyze the 19 metrics in the Fig. 3, we can draw that the UMD can not only unify the method's descriptor, but also improve the functional cohesiveness, decrease the degree of coupling and increase the sum or the average of the complexity of all the methods in a class.

For the metric *LOC*, we can see the code line increased by 80~200% drastically and this data is very high at first glance. However, the code instructions only accounts for a small part of the class file, and the most places is used to store the data of constant pool. And then, the *LOC* metric has small influence on the average size of class file as is shown in the Fig. 2(a).

IV. Conclusions

This paper propose a new obfuscation method UMD to smooth out the differences among methods' descriptor. The preceding obfuscation methods ignore the parameter types and the return value type existed in *MethodDescriptor*, which can help attacker rebuild a program easier. The UMD strives to modify parameter types and return value type to a uniform format, eliminate the information existed in *MethodDescriptor*, increase the complexity on the reverse engineering. With the tool of *CKJM extended*, we may safely draw the conclusion that in addition to unify the method descriptor, the UMD can also strengthen the cohesion in methods, improve the average complexity, reduce the coupling code and the like.

Other possible future work based on UMD is to implement a control flow obfuscation in a new way. Considering the method's descriptor has been unified, the main difficulty during the method invocation transforming is sharply reduced. As a result, we can have more flexibility to select more efficient obfuscation algorithm.

Acknowledgments

This work is supported by NSFC (Grant Nos. 61300181, 61502044), the Fundamental Research Funds for the Central Universities (Grant No. 2015RC23).

References

[1] Chan J T, Yang W. Advanced obfuscation techniques for Java bytecode[J]. Journal of Systems & Software, 2004, 71(1-2):1-10.

[2] Hou T W, Chen H Y, Tsai M H. Three control flow obfuscation methods for Java software[J]. IEE Proceedings - Software, 2006, 153(2):80-86.

[3] Majumdar A, Thomborson C, Drape S. A Survey of Control-Flow Obfuscations.[C]// International Conference on Information Systems Security. 2006:353-356.

[4] Foket C, De Sutter B, De Bosschere K. Pushing Java Type Obfuscation to the Limit[J]. IEEE Transactions on Dependable & Secure Computing, 2014, 11(6):553-567.

[5] Lindholm T, Yellin F, Bracha G, et al. The Java Virtual Machine Specification, Java SE 7 Edition[C]// Addison-Wesley Professional, 2013:27-59.

[6] Bodden E, Sewe A, Sinschek J, et al. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders[C]// International Conference on Software Engineering. IEEE, 2011:241-250.

[7] Blackburn S M, Garner R, Hoffmann C, et al. The DaCapo benchmarks: java benchmarking development and analysis[J]. Acm Sigplan Notices, 2006, 41(10):169-190.

[8] Diomidis Spinellis. Tool writing: A forgotten art? IEEE Software, 22(4):9–11, July/August 2005. (doi:10.1109/MS.2005.111).

[9] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," Software Engineering, IEEE Transactions on, vol. 20, pp. 476-493, 1994.

[10] Bansiya J, Davis C G. A hierarchical model for object-oriented design quality assessment[J]. IEEE Transactions on Software Engineering, 2002, 28(28):4-17.