18

Theory and Practice of Program Obfuscation

Xuesong Zhang, Fengling He and Wanli Zuo
JiLin University
China

1. Introduction

Software piracy has long been a confusing challenge to the software industry; especially with the popularity of the Internet today, this threat is growing more seriously. As a valuable form of data, software represents significant intellectual property. However, reverse engineering of software code by competitors may reveal important technological secrets, bring great harm to software developers and software providers. Because of this deterministic and self-cleared behaviour, as well as the environmental dependency property, when running under a malicious host, software may be accessed and modified by infinite resources and tools, all useful information would be definitely exposed to the attacker, which brings about great difficulty to software protection.

Along with the intensification of software market competition, technology theft poses another threat to the intellectual property rights protection. The competitors may analyze and collect the key technology or algorithm in the software through reverse engineering, which will quickly narrow the technology gap. They can also adjust their strategy according to the weakness or leakage explored from the software, and then they can use them to carry on some attacks, resulting in malicious competition. In some cases, the competitors may even do not need to understand the software internal working principle, they can directly extract the key code and integrated it into their own software to effectively enhance their competitiveness, thus seize the market share.

Clearly, there is a strong need for developing more efficient and effective mechanisms to protect software from becoming the victim of reverse engineering. Among those major approaches developed by different researchers, program obfuscation seems to be one of the most promising techniques. The concept of obfuscation was first mentioned by Diffie and Hellman (1976). When introducing the public-key cryptosystem, they claimed that, given any means for obscuring data structures in a private-key encryption scheme, one could convert this algorithm into a public-key encryption scheme.

Informally, obfuscation is a kind of special translation process. It translates a “readable” program into a function equivalent one, but which is more “unreadable” or harder to understand relatively. This kind of translation has the widespread potential applications both in cryptography and software protection, such as designing homomorphic public-key cryptosystems, removing random oracles from cryptographic protocols and converting private-key encryption schemes into public-key ones etc. in cryptography, or preventing reverse engineering (Collberg et al. (1997, 1998a, 1998b)), defending against computer viruses (Cohen (1993), Josse (2006)), protecting software watermarks and fingerprints

Source: Convergence and Hybrid Information Technologies, Book edited by: Marius Crisan,
ISBN 978-953-307-068-1, pp. 426, March 2010, INTECH, Croatia, downloaded from SCIYO.COM

(Collberg & Thomborson (2000), Naccache et al. (1999)) and providing security of mobile agents (D'Anna et al. (2003), Hohl (1998)) etc. in software protection. The main difference between these two research directions is: the former is based on information theory, its goal is to try to get an obfuscator with well-defined and provable security, while the later is based on software engineering, though lacking the firm ground for estimating to what extent such methods serve the purpose, it does increase program complexity, bring barriers to program understanding.

The chapter is structured as follows: Section 2 reviews various forms of formal definition of obfuscators put forward by different researchers and the corresponding positive and negative effect on the possibility of such obfuscator. Part of this section refers to the survey made by Wyseur (2009). Section 3 gives a systemic description of software obfuscation based on software engineering perspective, it is mainly composed of the concept and taxonomy developed by Collberg et al. (1997), and it also includes some most recently new research results. In Section 4 and Section 5, we propose two Java program obfuscating method respectively, namely the call-flow obfuscation and instruction obfuscation. Section 6 concludes the paper.

2. Obfuscation theory

2.1 Notations

PPTdenotes probabilistic polynomial-time Turing machine. For $PPT\ A$ and any input x the output $A(x)$ is a random variable. $A^M(x)$ denote the out put of A when executed on input x and oracle access to M .We will write $|A|$ to denote the size of A . For a pair of Turing machines A and B , $A \approx B$ denotes their equivalence, i.e. $A(x) = B(x)$ holds for any input x .Function $f : N \rightarrow [0,1]$ is negligible if it decreases faster than any inverse polynomial, i.e. for any $k \in N$ there exists n_0 such that $f(n) < 1/n^k$ holds for all $n \geq n_0$. We use $neg(\cdot)$ to denote unspecified negligible function.

2.2 Definitions of obfuscation

The first contributions towards a formalization of code obfuscation were made by Hada (2000), who presented definitions for obfuscation based on the simulation paradigm for zero knowledge. The main difference between the obfuscation definition and the simulation-based definition used in (black-box) cryptography, lies in the type of objects the adversary interacts with. In the obfuscation case, it is a comparison between (white-box) interaction to an implementation of the primitive, and the interaction with an oracle implementation (black-box). In the tradition cryptography case, it is between an oracle implementation of the cryptographic primitive, and an idealized version. This new concept is captured by the Virtual Black-Box Property (VBBP). Informally, obfuscators should satisfy the following two requirements: (1) functionality: the new program has the same functionality as the original one and (2) Virtual Black-Box Property: whatever one can efficiently compute given the new program, can also be computed given oracle access to the original program. The functionality requirement is a syntactic requirement while the virtual black-box property represents the security requirement that the obfuscated program should be unintelligible. The definition of obfuscation was firstly formalized by Barak et al. (2001).

Definition 1 (Obfuscator): A probabilistic algorithm O is an obfuscator if the following three conditions hold:

- Functionality: $\forall P \in \mathcal{P}$, $O(P)$ has the same function as P.
- Polynomial slowdown: There is a polynomial p, such that for every P, $|O(P)| \leq p(|P|)$, and if P halts in t steps on some input x, then $O(P)$ halts in $p(t)$ steps on input x.
- Virtual Black-Box Property: Given access to the obfuscated program $O(P)$, an adversary should not be able to learn anything more about the program P, than it could learn from oracle access to P.

The Virtual Black-Box Property was defined in several different notions by Barak et al. (2001).

Predicate-based obfuscation

In this notion, an adversary aims to compute some predicate on the program P. In this sense, the virtual black-box property captures that for any adversary and any Boolean predicate π, the probability that an adversary is able to compute $\pi(P)$ given the obfuscation $O(P)$ should be comparable to the probability that a simulator S is able to compute $\pi(P)$ when given only oracle access to P. Roughly speaking, this guarantees that the adversary A does not have any advantage of white-box access, compared to a black-box simulation, hence the obfuscation does not leak any extra information on $\pi(P)$.

Definition 2 (Predicate-based Virtual Black-Box Property): An obfuscator O satisfies the Predicate-based Virtual Black-Box Property if for any predicate π and for any (polynomial time) adversary A, there exists a (polynomial time) simulator S, such that for $\forall P \in \mathcal{P}$:

$$|\Pr[A(1^{|P|}, O(P)) = \pi(P)] - \Pr[S_A^P(1^{|P|}) = \pi(P)]| \leq neg(|P|),$$

where the probabilities are taken over the coin tosses of A, S, and O.

As pointed out by Barak et al. (2001) and Hohenberger et al. (2007), the predicate definition does give some quantifiable notion that some information (i.e., predicates) remains hidden, but other non-black-box information might leak and compromise the security of the system. This lead to a stronger notion of "virtual black-box".

Distinguisher-based obfuscation

This notion of obfuscation is based on computational indistinguishability, and does not restrict what the adversary is trying to compute. For any adversary given the obfuscated program $O(P)$, it should be possible to construct a simulator S (with only oracle access to P) that is able to produce a similar output. This notion of similarity is captured by a distinguisher D.

Definition 3 (Distinguisher-based Virtual Black-Box Property): An obfuscator O satisfies the distinguisher-based Virtual Black-Box Property if for any (polynomial time) adversary A, there exists a (polynomial time) simulator S, such that that for $\forall P \in \mathcal{P}$:

$$|\Pr[D(A(O(P))) = 1] - \Pr[D(S^P(1^{|P|})) = 1]| \leq neg(|P|),$$

where D is a distinguisher, and the probabilities are taken over the coin tosses of A, S, and O.

This notion of security is quite similar to the notion of semantic security for (black-box) cryptographic schemes. As pointed out by Wee (2005), this removes the need to quantify over all adversaries, as it is necessary and sufficient to simulate the output of the obfuscator. To avoid trivial obfuscation, Hofheinz et al. (2007) extended the distinguisher-based definition by giving the distinguisher oracle access to the functionality P . This leads to a very strong notion of obfuscation.

The above definitions are defined for cryptography purpose, however, for most program obfuscation in real world, this virtual black-box condition is too strong. For ordinary software, it is usually supplied with a user manual specifying its functionality. That is, the adversary knows the function the program compute. The aim of obfuscation in this case is not to hide any property of the program which refers to its functionality, but to make unintelligible the implementation of these functional properties in a particular program. This lead to a non black-box definition − Best-possible obfuscation (Goldwasser & Rothblum (2007)).

Best possible obfuscation

Best possible obfuscation makes the relaxed requirement that the obfuscated program leaks as little information as any other program with the same functionality (and of similar size). In particular, this definition allows the program to leak non black-box information. Best-possible obfuscation guarantees that any information that is not hidden by the obfuscated program is also not hidden by any other similar-size program computing the same functionality, and thus the obfuscation is (literally) the best possible.

Definition 4 (Distinguisher-based Best-possible Obfuscation): An obfuscator O is said to be a best possible obfuscator if there exists a (polynomial time) simulator S , such that for any two programs $P_1, P_2 \in \mathcal{P}$ that compute the same function, and $|P_1|=|P_2|$, such that:

$$|\Pr[D(O(P_1))=1]-\Pr[D(S(P_2))=1]| \leq neg(|P_1|),$$

where D is a distinguisher, and the probabilities are taken over the coin tosses of S and O .

Instead of requiring that an obfuscator strip a program of any non black-box information, this definition requires only that the (best-possible) obfuscated program leak as little information as possible. Namely, the obfuscated program should be "as private as" any other program computing the same functionality (and of a certain size). A best-possible obfuscator should transform any program so that anything that can be computed given access to the obfuscated program should also be computable from any other equivalent program (of some related size). A best-possible obfuscation may leak non black-box information (e.g. the code of a hard-to-learn function), as long as whatever it leaks is efficiently learnable from any other similar-size circuit computing the same functionality.

While this relaxed notion of obfuscation gives no absolute guarantee about what information is hidden in the obfuscated program, it does guarantee (literally) that the obfuscated code is the best possible. It is thus a meaningful notion of obfuscation, especially when we consider that programs are obfuscated every day in the real world without any provable security guarantee. In this sense, it may be conjectured that best possible obfuscation is more closed to software protection obfuscation.

Apart from these three definitions above, there are other notions of obfuscation, such as that based on computational indistinguishability, satisfying a relation or computing a predicate, we refer Barak et al. (2001), Hofheinz et al. (2007), Hohenberger et al. (2007), Kuzurin et al.(2007), Wee (2005) for more details.

2.3 Negative results

In their seminal paper, Barak et al. (2001) show that it is impossible to achieve the notion of obfuscation according to Definition 2, that is, it is impossible to construct a generic obfuscator for all family of programs P. This is proved by constructing a family of functions F which is inherently unobfuscatable in the sense that there exists some predicate $\pi : F \to \{0,1\}$ that can be computed efficiently when having access to an obfuscated implementation $O(f)$ of $f \in F$, but no efficient simulator can compute $\pi(f)$ much better than by random guessing, given solely oracle access to f. This result follows from the following paradox.

- If one-way functions exist, then there exists an inherently unobfuscatable function ensemble.
- The existence of an efficient obfuscator implies the existence of one-way functions.

As a result of the above, it can be concluded that efficient obfuscators do not exist.

Due to the paradox, every cryptographic primitive that implies the existence of a one-way function, implies the existence of a respectively unobfuscatable primitive. This applies to digital signature schemes, symmetric-key encryption schemes, pseudo-random function ensembles, and MAC algorithms.

Goldwasser and Kalai (2005) argued in the predicate-based notion of obfuscation to hold in the presence of auxiliary input. They observe that this is an important requirement for many applications of obfuscation, because auxiliary input comes into play in the real world. They prove that there exist many natural classes of functions that cannot be obfuscated with respect to auxiliary input (both dependent and independent auxiliary input).

Wee (2005) explored obfuscation of deterministic programs under the strong (distinguisher-based) notion of obfuscation, and concluded that deterministic functions can be obfuscated if and only if the function is learnable. Hofheinz et al. (2007) also remarked that any family of deterministic functions must be approximately learnable to be obfuscatable (in their augmented strong notion of obfuscation). Hence, it is not possible to obfuscate (deterministic) pseudo-random functions under their definition.

On non black-box definition, Goldwasser and Rothblum (2007) show that if there exist (not necessarily efficient) statistically secure best-possible obfuscators for the simple circuit family of 3-CNF circuits, then the polynomial hierarchy collapses to its second level, and give the impossibility result for (efficient) computationally best-possible obfuscation in the (programmable) random oracle model.

2.4 Positive results

A positive result on obfuscation was presented prior to the first formulation of definitions for obfuscation. Canetti (1997) presented a special class of functions suitable for obfuscation under very strong computational assumptions, that works for (almost) arbitrary function distributions. In subsequent work, Canetti et al. (1998) presented a construction suitable for obfuscation under standard computational assumptions, which is proved secure for uniform function distribution. Both results are probabilistic and technically very sophisticated.

Lynn et al. (2004) explored the question of obfuscation within an idealized setting − the random oracle model, in which all parties (including the adversary) can make queries to a random oracle. The heart of their construction is the obfuscation of a point function. A point function $I_\alpha(x)$ is defined to be 1 if $x = \alpha$, or 0 otherwise, and they observed that in the random oracle model point functions can be obfuscated, leading to obfuscation algorithms for more complex access control functionalities. Under cryptographic assumptions, it is also known how to obfuscate point functions without a random oracle. Canetti (1997) showed (implicitly) how to obfuscate point functions (even under a strong auxiliary-input definition), using a strong variant of the Decisional Diffie-Hellman assumption. Wee (2005) presented a point function obfuscator based on the existence of one-way permutations that are hard to invert on a very strong sense. Wee also presented a construction for obfuscating point functions with multi-bit output, which are point functions $I_{\alpha,\beta}(x)$ that evaluate to β on input α , and to 0 on any other input.

Most of the obfuscation definitions presented above, are either too weak for or incompatible with cryptographic applications, have been shown impossible to achieve, or both. Hohenberger et al. (2007) and Hofheinz et al. (2007) present new definitions which have a potential for interesting positive results. Hohenberger et al. introduce the notion of average-case secure obfuscation, based on a distinguisher-based definition that allows families of circuits to be probabilistic. They present a probabilistic re-encryption functionality that can be securely obfuscated according to this new definition. Similarly, Hofheinz et al. present another variant of a distinguisher-based definition. The deviation is that they consider probabilistic functions and select the function to be obfuscated according to a distribution. Their new notion is coined average obfuscation. The goal is to consider obfuscations for specific applications, and they demonstrated the obfuscation of an IND-CPA secure symmetric encryption scheme that results into an IND-CPA secure asymmetric scheme. Similar results hold for the obfuscation of MAC algorithms into digital signature schemes.

3. Heuristic obfuscation

Despite the fundamental results so far from theoretical approaches on code obfuscation, their influence on software engineering of this branch is minor: security requirements studied in the context of cryptographic applications are either too strong or inadequate to many software protection problems emerged in practice. Everybody dealing with program understanding knows that, in many cases, even small programs require considerable efforts to reveal their meaning. This means that there exists the possibility of some weakly secure obfuscators. Program obfuscation is received more attention gradually exactly based on this viewpoint in software engineering in the last decade. The practical goal of obfuscation is then to make reverse engineering uneconomical by various semantic preserving transformations, it is sufficient that the program code be difficult to understand, requiring more effort from the attacker than writing a program with the same functionality from scratch.

Early attempts of obfuscation aim at machine code level rewriting. Cohen (1993) used a technique he called "program evolution" to protect operating systems that included the replacement of instructions, or small sequences of instructions, with ones that perform semantically equal functions. Transformations included instruction reordering, adding or removing arbitrary jumps, and even de-inlining methods. However, it was until the appearance of the paper by Collberg et al. (1997), software engineering community became acquainted with obfuscation. They gave the first detailed classification of obfuscating transformations together with the definition of some analytical methods for quality measures.

3.1 Types of obfuscation

Lexical obfuscation

This involves renaming program identifiers to avoid giving away clues to their meaning. Since the identifiers have little semantic association with program itself, their meaning can be inferred from the context by a determined attacker, lexical obfuscation has very limited capability and is not alone sufficient. Typical Java obfuscators such as SourceGuard[1] and yGuard[2] etc. all implement this kind of obfuscation. It is worth noting that, in order to mislead the analyzer, Jaurora[3] randomly exchange identifiers instead of scramble them, thus, has the more secrete property. Chan et al. (2004) 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 bytecode level by some automated tools (Cimato et al. (2005)).

Data obfuscation

This transformation targets at data and data structures contained in the program, tries to complicate their operations and obscures their usage, such as data encoding, variable and array splitting and merging, variable reordering, and inheritance relation modifying. Among them, the array reconstruction method receives more attention in recent years. Array splitting, merging, folding, and flattening was discussed by Collberg (1998a) in detail. Further researches are also carried out later, such as generalized array splitting method (Drape (2006)), composite function based indexing method(Ertaul & Venkatesh (2005)), homomorphic function based indexing method (Zhu (2007)), and class encapsulated array reconstruction method (Praveen & Lal (2007)) etc. Data obfuscation is especially useful for protecting object-oriented application since the inheritance relation is crucial to software architecture undestanding. Sonsonkin et al. (2003) present a high-level data transformations of Java program structure − design obfuscation. They replaced several classes with a single class by class coalescing, and replaced a single class with multiple classes by class splitting. They hold that, 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.

Control Obfuscation

This approach alters the flow of control within the code, e.g. reordering statements, methods, loops and hiding the actual control flow behind irrelevant conditional statements. This form of obfuscation can be further divided into two categories, dynamic dispatch and opaque predicate. For the dynamic dispatch, Wang et al. (2001) proposed a dynamic dispatch model based on the fact that aliases in a program drastically reduce the precision of static analysis of the program. Chow et al. (2001) transformed a program to flat model by dividing it into basic blocks, and embed into it an intractable problem with respect to computational complexity theory. Consequently, to determine the target address is equivalent to solving the intractable problem. Toyofuku et al. (2005) assigned each method with a unique ID. During program execution, the control flow will randomly points to any method, and whether the target method will execute or not is based on the comparison

[1] http://www.4thpass.com/

[2] http://www.yworks.com/products/yguard/

[3] http://wwwhome.cs.utwente.nl/~oord/

between the method's ID and a global variable that is updated after each method execution. An opaque predicate is a conditional expression whose value is known to the obfuscator, but is difficult for an adversary to deduce statically. For the construction of opaque predicate, Collberg's (1998b) algorithm is based on the intractability property of pointer alias, Venkatraj's (2003) algorithm is based on well-known mathematical axioms and probability distribution, Palsberg's (2000) algorithm is based on data structure correlation, while Drape's (2007) algorithm is based on program slicing information.

Prevention obfuscation

This transformation is quite different in flavor from control or data transformation. In contrast to these, their main goal is not to obscure the program to human reader. Rather, it is designed to make known automatic deobfuscation techniques more difficult, or to explore known problems in current deobfuscators or decompilers, e.g. junk bytes insertion. Some dynamic dispatching methods inherently have this capability. Batchelder and Hendren (2007) proposed a number of prevention transformation techniques for Java program by exploiting the semantic gap between what is legal in source code and what is legal in bytecode. The methods include converting branches to jsr instructions, disobeying constructor conventions, and combining try blocks with their catch blocks etc., all which lead to the decompilers failing to decompile the bytecodes. Instead of obfuscating the program itself, Monden et al. (2004) gave an idea for obfuscating the program interpretation. If the interpretation being taken is obscure and thus it can not be understood by a hostile user, the program being interpreted is also kept obscure since the user lacks the information about "how to read it."

3.2 Quality of obfuscation

According to Collberg (1997), there are four main metrics measure the effectiveness of an obfuscating transformation in terms of potency, resilience, cost, and stealth. Potency measures the complexity added to the obfuscated program. Resilience measures how well the transformation holds up under attack from an automatic deobfuscator. Cost measures the execution time/space penalty of obfuscating a program. Stealth measures how much the obfuscated code looks like the original one and how well it fits in with the other code. These proposed measures are known as analytical methods, since they extract information by taking obfuscation algorithms parameter, source program and obfuscated program. Utilizing these metrics, Wroblewski (2002) gave a thorough comparison of different obfuscation algorithms based on source code level transformation. Dyke and Colin (2006) proposed an obfuscation method at assembly-code level and did a similar comparison work. Karnick et al. (2006) developed an analytical method based on these metrics to evaluate the strength of some commercial Java obfuscators.

The drawback of these metrics is that they do not define exactly to what extent the difficulty or hardness it takes to understand the obfuscated program compared to the original one for an analyzer. This is partially due to the considerable gap between theory and practice of program obfuscation (Kuzurin et al. (2007)). The formal definition of obfuscation for cryptography purpose is not suitable for most program protection applications in real world. Thus, how to clearly reveal the most important common properties required in any software obfuscation and give the corresponding effective measure metrics still need a long road to run.

4. Call-flow obfuscation

Linn and Debray (2003) proposed the concept of branch function in their native code tamper-proofing algorithm. Unconditional branch instructions are converted to calls to a branch function. The branch function will transfer execution to the original target based on information of stack, which will prevent a disassembler from identifying the target address, thus resist to static analysis effectively. Unfortunately, this algorithm is only applicable to native code which can access and modify its own stack, but not suitable to Java byte code. As a control obfuscation algorithm, the proposed scheme generalized this idea and apply it to Java object-oriented language. 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. This kind of obfuscation algorithm is called as call-flow obfuscation. In Fig. 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 up cast to their common base type, which object's *DoIt* method will really execute can only be ascertained at runtime. Static analyze of this kind of single level type with dynamic dispatch inter-procedure points-to is PSPACE hard (Chatterjee et al. (2001)).

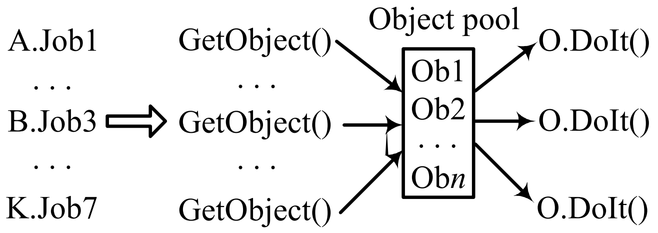


Fig. 1. The object pool model

4.1 Obfuscation algorithm

In Java language, 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 section, obfuscation scope only refers to those packages developed by the programmer himself.

The obfuscation algorithm mainly consists of three steps, namely the invocation format unification, inter-classes method merging, and object pool construction.

Invocation format unification

The parameters and return types defined in a method of a program are usually different from each other. In order to perform inter-classes method merging, their invocation formats should be unified. The two classes import in Fig. 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. In this case, there are three flags in *ParamObject*, which means 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 be eventually embedded into the *DoIt* method of some subclasses inherited from *DoJob*.

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

Fig. 2. Unification of invocation format

```
public class A {                                    public class B {
  public int DoJobA1(int x)                           public int DoJobB1(int x, int y)
  public long DoJobA2(double x, double 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');
}
```

Fig. 3. The 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;
  }
}
```

Fig. 4. Inter-classes method merging

Inter-classes method merging

In determining which method can be merged, 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
- Methods whose internal codes invoke other non-public methods, which inherited from super class or super interface that is outside of the obfuscation scope.

Fig. 4 shows a possible merging instance of two classes defined in Fig. 3. Method *DoJobA1* and *DoJobB2* which belong to class *A* and class *B* respectively are merged into one *DoIt* method. Since it only needs two flags in this instance, other flags 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 not destroy the data or control flow of the program.)

When carry on method merging, three special situations need handling.

Method polymorphism: If the merged method is inherited from super class or super interface that is within 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 methods defined in current class or its super class, eg. an invocation to *DoJobA2* inside method *DoJobA1*. There are two approaches to this situation:

- If *DoJobA2* is a user-defined method, it can be merged further, otherwise, its access property is modified to public, and the following action is taken the same as the second approach.
- The invocation is transformed to the standard form by adding a qualifier in front of the invocated method, i.e. *DoJobA2* is converted to *a. DoJobA2*. The qualifier *a* is an instance of class *A* which is put into the object array of *ParamObject* as an additional parameter.

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

- Class qualifier can be added before the fields accessed by this method, which is similar to the second approach in method dependency. But this is not suitable for the non-public field inherited from super class that is outside of the obfuscation scope.
- This solution adds *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.

Object pool construction

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. The universal hashing is a desired candidate to construct the object pool here.

The main idea behind universal hashing 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. Due to this randomization property, the algorithm can behave differently on each execution, even for the same input, guaranteeing good average-case performance for any input.

Definition 5: Let H be a finite collection of hash functions that map a given universe U of keys into the range $\{0,1,\cdots,m-1\}$. Such a collection is said to be universal if for each pair of distinct keys $k,l \in U$, the number of hash functions $h \in H$ 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 $\{0,1,\cdots,p-1\}$, and let Z_{p2} denote the set $\{1,2,\cdots,p-1\}$. 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.

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

The universal hashing table is used 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 enables us to assign different values to a_1 and a_2 each time the program start. According to this characteristic, any expressions can be constructed based on a_1 and a_2, and be used as the key to access hashing table. In this case, the key is no longer a constant, and better information hiding result obtained. Fig. 5 shows the structure of hashing table. In which, 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 Fig. 6, invocation to class *A*'s mehod *DoJobA1* is replaced by invocation to *DoIt* method in one of *DoJob*'s subclass.

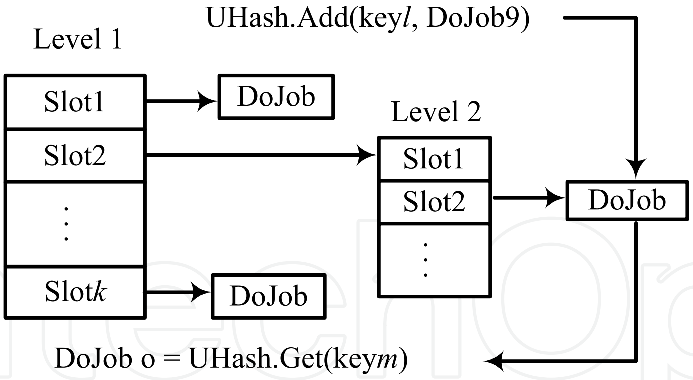
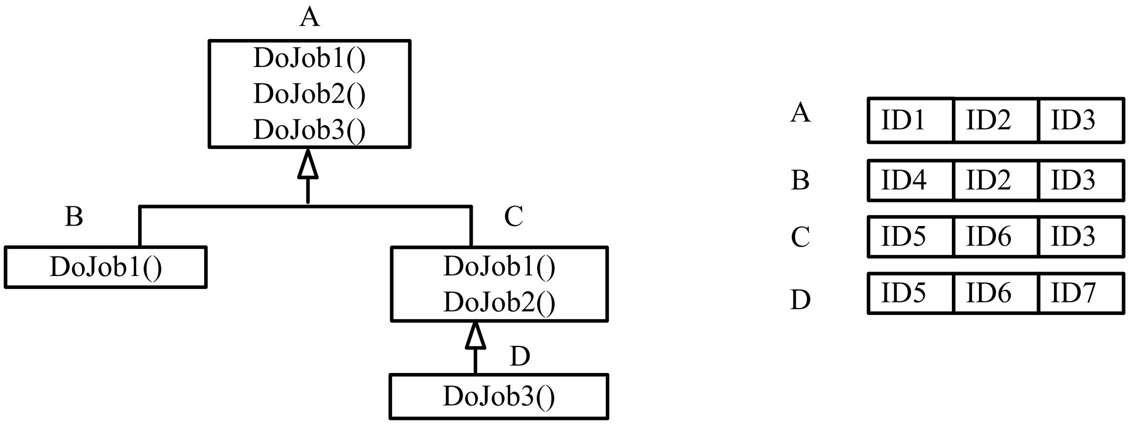


Fig. 5. Structure of hashing table

However, using key to access hashing table directly will cause some problem when in face of method polymorphism. Consider the inherited relation in Fig. 7, if all methods in class *A* are extracted and embedded into some *DoIt* methods, method extraction should be performed during each 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 of *a.DoJob1*. As a result, it still cannot be determined that which key should be used to access the hashing table. Under this situation, one method *GetIDs* should be appended 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. Fig. 7 shows the return arrays of each class corresponding to the left side. 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.

```
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);
```

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



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

Fig. 7. Method polymorphism and return array

4.2 Obfuscating enhancement

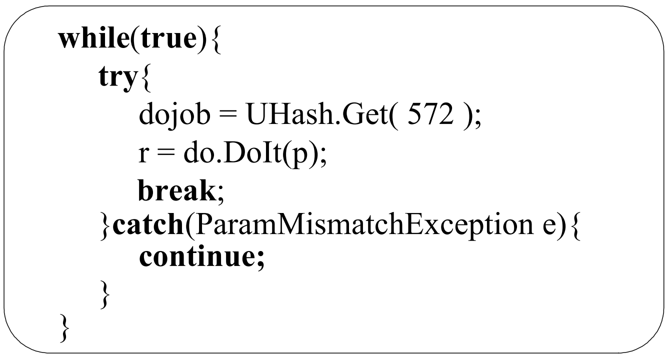
In order to perform effective attack, which instance each *DoJob* references to should be precisely located. Since all objects added into the object pool have been upcast to their super class *DoJob*, and different keys are used to store and access the same object in hashing table. It is not feasible to clarify all the call-flows in a program relying solely on static analysis. 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: This approach makes 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. Wenever a merged method is executed, the same functionality can be realized whatever object is selected.

Method interleaving: Since 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 further action can be taken to add some garbage codes which reference to these fields.

Hashing table extension: This aproach extends some slots to insert new objects. The parameters used in *DoIt* method of these newly inserted object are different from the *DoIt* method of objects that have already existed. When a slot that includes more than one objects is located by the given key, the return object is randomly selected from those in this slot. Before entering 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 (Fig. 8), and following instructions will not be executed until a success invocation to the *DoIt* method.



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

Fig. 8. The *DoIt* method invocation model after hashing table extension

Dynamic adjusting of object pool: Multi-rules can be adapted simultaneously to construct the object pool. At the program start point, one operation rule is randomly selected, and a new thread is introduced 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.

4.3 Experimental result

We extend the refactor plugin in Eclipse, and apply this scheme to five java programs. Currently, only the basic obfuscating method is implemented, excluding those enhanced mechanisms such as multi-duplication, method interleaving etc. Some of the programs are Java applets which will never terminate without user interference. Their execution time is only measured in finite cycles. For 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.

Table 1 indicate that, the program size increasing ratio after obfuscating lies between 1.11 and 1.62. With the size growing of the original program, the ratio presents a downward trend. The reason lies in 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 starts. Once accessed, the object pool 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, the proposed scheme has little performance influence on the original program.

Program	Description	Method Merged		Before Obf.	After Obf.	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	22	Jar file size (byte)	59030	68555	1.17
			Execution time (sec)	8.71	9.15	1.05

Table 1. Experimental result by using only the basic obfuscation method

5. Instruction obfuscation

The concept of obfuscated interpretation was motivited by Monden et al. (2004). They employed a finite state machine (FSM) based interpreter to give the context-dependent semantics to each instruction in a program, thus, attempts to statically analyze the relation between instructions and their semantics will not succeed. In fact, our proposed method is similar to this technique. However, the FSM interpretation unit is hardware implemented, its state transition rule cannot change any more once being embedded. Further more, in order to maintain the same state at the top of any loop body in a translated program, a sequence of dummy instructions must be injected into the tail of the loop. These dummy instructions will destroy the correctness of the stack signature, which means it is very hard (if not impossible) to implement the translation of a Java program after which the translated program can still runs normally. In our scheme, the mapping rule is more generally defined, which can be easily changed at will. Because there is no need to insert dummy instructions, it is extremely easy to make a translated program looks like a normal program whether by reverse engineering or runtime inspection.

The core idea of this framework is to construct an interpreter *W*, which carries out obfuscated interpretations for a given program *P*, where *P* is a translated version of an original program P_0 written in Java bytecode. The obfuscated interpretation means that an interpretation for a given instruction *c* is not fixed; specifically, the interpretation for *c* is determined not only by *c* itself but also by other auxiliary input to *W* (Fig. 9).

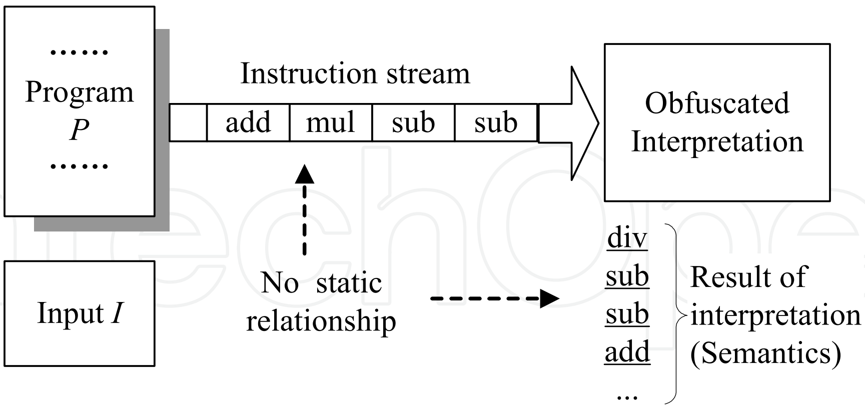


Fig. 9. Obfuscated interpretation concept

In order to realize the obfuscated interpretation in *W*, a permutation mechanism is employed that takes input as an instruction *c* where each permutation makes a different interpretation for *c*. Since the interpretation for a particular type of instruction varies with respect to permutation definitions, we call such *W* a permutation-based interpreter. In this framework, *W* is built independent of P_0; thus, many programs run on a single interpreter *W*, and any of the programs can be easily replaced to a new program for the sake of updating.

5.1 Framework for obfuscated interpretation

Overview

The following diagram (Fig. 10) shows brief definitions of materials related to *W*.

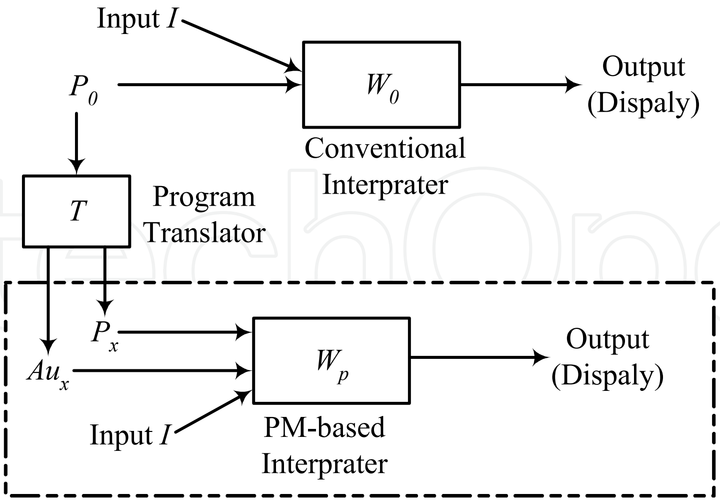


Fig. 10. Obfuscated interpretation framework

P_0**:** is a target program intended to be hidden from hostile users. Let us assume that P_0 is written in bytecode, where each statement in P_0 consists of a single opcode and (occasionally) some operands.

W_0**:** is a common (conventional) interpreter for P_0 such as a Java Virtual Machine.

P_x**:** is a "translated program" containing encoded instructions whose semantics are determined during execution according to the auxiliary input Au_x. This P_x is an equivalently translated version of P_0, i.e., P_x has the same functionality as P_0.

I**:** is an input of P_0 and P_x. Note that P_0 and P_x take the same input.

W_p**:** is a permutation-based interpreter that can evaluate encoded instructions of P_x according to the auxiliary input Au_x. This W_p is an extension of W_0 with a permutation unit. Each W_p has a unique key which is used to decrypt Au_x.

T**:** is a program translator that automatically translates P_0 into P_x with respect to the randomly created one-to-many map among all possible instructions.

Au_x: is the one-to-many mapping rule that describe how one opcode map to the others generated randomly by T when translating a program. The content of Au_x is encrypted by the same key as that of W_p.

In this framework, it is assumed that W_p is hidden from the end user as much as possible, e.g., it is integrated with the Java Virtual Machine. However, P_x must be delivered to the user and put in an accessible area so as to enable it to update. Each W_p should be equipped with a different key so that an adversary cannot easily guess one W_p's interpreter after having "cracked" some other W_p's interpreter.

Permutation Unit

The permutation unit denoted as W_p can be defined as follows:

$\sum = \{c_0, c_1, \ldots, c_{n-1}\}$ is the input alphabet.

$\Psi = \{\underline{c_0}, \underline{c_1}, \ldots, \underline{c_{n-1}}\}$ is the output alphabet.

$\prod = \{\pi_0, \pi_1, \ldots, \pi_{n-1}\}$ is the auxiliary input alphabet. It is decrypted from Au_x using the key owned by W_p.

$\lambda_i : \sum \times \prod \rightarrow \Psi$ is the output function.

$\Lambda = (\lambda_0, \lambda_1, \ldots, \lambda_{n-1})$ is the n -tuple of all output functions. Λ is the specification of a W_p that defines a dynamic map between obfuscated instructions.

Based on Monden et al. (2004), there are four types of design choices for the interpreter, which are dependent upon the instruction set used for P_x. Let $InsP_0$ and $InsP_x$ be the instruction sets for P_0 and P_x,. In Type 1 design, the instruction set for P_x is the same as that for P_0, so $InsP_x = InsP_0$. In the rest of this section, we focus on this design type. Let us assume $InsPx = \sum \cup O$ where elements $c_i \in \sum$ are obfuscated instructions, and $o_i \in O$ are non-obfuscated instructions. This means, P_x contains both c_i and o_i, and, if the permutation unit recognizes $c_i \in \sum$ as input then its semantics is determined by the auxiliary input π_i, otherwise an input $o_i \in O$ is directly passed to the execute unit. Each underlined symbol c_i in Ψ denotes the normal (untranslated) semantics for the correspondingly-indexed opcode c_i in $\sum$.

Here is a simple example of W_p where

$\sum = \{iadd, isub, imul, idiv\}$

$\Psi = \{iadd, isub, imul, idiv\}$

$\prod = \{0, 1, 2, 3\}$

$\Lambda = (\lambda_0(iadd, 0) = \underline{isub}, \lambda_0(isub, 0) = \underline{isub}, \lambda_0(imul, 0) = \underline{iadd}, \lambda_0(idiv, 0) = \underline{imul}, \lambda_1(iadd, 1) = \underline{idiv}, \lambda_1(isub, 1) = \underline{imul}, \lambda_1(imul, 1) = \underline{idiv}, \lambda_1(idiv, 1) = \underline{isub}, \lambda_2(iadd, 2) = \underline{iadd}, \lambda_2(isub, 2) = \underline{iadd}, \lambda_2(imul, 2) = \underline{isub}, \lambda_2(idiv, 2) = \underline{idiv}, \lambda_3(iadd, 3) = \underline{imul}, \lambda_3(isub, 3) = \underline{idiv}, \lambda_3(imul, 3) = \underline{imul}, \lambda_3(idiv, 3) = \underline{iadd})$

This W_p takes an encoded opcode $c_i \in \{iadd, isub, imul, idiv\}$ as an input, translates it into its semantics (cleartext opcode) $\underline{c_i} \in \{\underline{iadd}, \underline{isub}, \underline{imul}, \underline{idiv}\}$ according to π_i, and outputs $\underline{c_i}$. Fig. 11

shows an example of interpretation for an instruction stream given by this W_p. Obviously, even this simple permutation has the ability to conduct an obfuscated interpretation.

c_i	π_i		$\underline{c}_i$
iadd	3		imul
imul	0		iadd
idiv	2	⟶	idiv
imul	1		idiv
isub	2		iadd

Fig. 11. Instruction stream interpretation

When concerning the other design types, the input alphabet $\sum$ and the auxiliary input alphabet $\prod$ is larger, which will eventually resulting to a more complex W_p.

Program Translator

The translator T can be defined based on interpreter W_p:

$\sum' = \Psi = \{\underline{c}_0, \underline{c}_1, \ldots, \underline{c}_{n-1}\}$ is the input alphabet.

$\Psi' = \sum = \{c_0, c_1, \ldots, c_{n-1}\}$ is the output alphabet.

$\prod = \{\pi_0, \pi_1, \ldots, \pi_{n-1}\}$ is the auxiliary output alphabet. It is encrypted by the key of W_p to get Au_x.

$\lambda'_i : \sum' \rightarrow \Psi' \times \prod$ is the output function.

$\Lambda' = (\lambda'_0, \lambda'_1, \ldots, \lambda'_l)$, is a tuple of all possible output functions. Its item count l is determined by the permutation rule, and is larger than n by far.

The definition above shows that λ'_i is a one-to-many function. This non-deterministic characteristic is the key idea to make each translation of P_0 different from the other.

In order to make P_x pass Java's bytecode verifier, bytecode that can substitute each other without causing any stack or syntactic errors must be classified into subgroups carefully according to their operand type and operand number. For example, the four instructions *iadd*, *isub*, *imul* and *idiv* belong to the same group, because they all pop two integers, perform relevant operation, and then push the integer result back into the stack. Each of them form a one-to-many relation to the others (including the instruction itself). Thus, the input (and output) alphabet is partitioned into many sub groups according to their features, such that symbols $c_0, c_1, \ldots, c_{a-1}$ are in the first group G_1 and the symbols $c_a, c_{a+1}, \ldots, c_{b-1}$ are in the second group G_2 , and so on.

During program translation, T only accepts those input instructions that belong to $\sum'$. For each accepted instruction, the following actions are performed:

- Decide the sub group G_j this instruction belongs to.
- Search for all $\lambda_k , \ldots , \lambda_{k+l} \in \Lambda$ that output $\underline{c}_i$ in G_j.
- Randomly select m, $k \quad m \quad k + l$, extract c_m and π_m from λ_m, and put them into Ψ' and $\prod$ respectively.

Fig. 12 shows an example of program translation corresponding to W_p of Fig. 11. As shown in Fig. 12, the output is considerably different from the input of Fig. 11. This means that given a program P_0, each time a different P_x is produced even for the same W_p. In other words, this framework can guarantee that each installed copy of a program is unique. More precisely, each installed copy differs enough from all other installed copies to ensure that successful attacks on its embedded copyright protection mechanism cannot be generalized successfully to other installed copies.

$\underline{c}_i$		c_i	π_i
<u>imul</u>		idiv	0
<u>iadd</u>		isub	2
<u>idiv</u>	⟶	isub	3
<u>idiv</u>		iadd	1
<u>iadd</u>		idiv	3

Fig. 12. Example of T: $P_0 \rightarrow P_x$

5.2 Implementation

We now consider some implementation issues of our framework on mobile phones. The platform of choice in this setting is Mysaifu JVM[4]. It is an open source Java Virtual Machine runs on the Windows mobile platform. It's runtime Java library is based on the GNU Classpath[5] whose target is to be fully compatible with the core of JDK1.5.

Auxiliary Input

When Mysaifu JVM opens a jar file, it loads all the class files into memory, and perform bytecode verification by iterating through all methods at the same time. JVM also load a few system classes such as JavaLangSystem, JavaLangString, JavaLangThread, JavaLangClass etc. immediately after startup. Then, the main class is executed by JVM interpreter. The best place to embed our Type 1 permutation unit into JVM is right in front of the verifier.

There are two places to store Au_x. One is in the jar manifest file as an extended option, the other one is in the class file itself as an attribute. In the former case, filename and relevant permutation rule must be included in Au_x, and when faced with incremental update, the correspondence between filename and permutation rule should also be updated, which will call for much more effort to do. In the latter case, we can replace any class file freely, without worry about the permutation rule. Because a Java virtual machine implementation is permitted to silently ignore any or all attributes in the attributes table of a code attribute, the attribute we added which include Au_x will not cause any error in a third party JVM. Here the latter one is the desired choice for this implementation.

In this target, SIM card number is used as the key to encrypt Au_x. When JVM find the given attribute in one code attribute table, it will decrypt the attribute info, and use this info to translate some instructions in the corresponding method. In this way, the obfuscated java software will be interpreted correctly in the modified JVM.

Experimental Result

We have implemented our framework by Visual Studio 2005 and Eclipse 3.2M.

The Java bytecode instructions are divide into two classes:

- Simple instructions (Table2): Instructions that can be substitute each other. They are classified into seven subgroups further. Subgroups that contain less than five instructions are omitted.
- Local storage related instructions (Table 3): In order to pass the check by bytecode verifier, a data-flow analysis is needed to guarantee that the local variable referenced by the submitted instruction has already been defined and initialized. Since the

[4] http://www2s.biglobe.ne.jp/~dat/java/project/jvm/

[5] http://www.gnu.org/software/classpath/

Subgroup	Instructions	Subgroup	Instructions
1	iconst_m1	4	fadd
	iconst_0		fsub
	iconst_1		fmul
	iconst_2		fdiv
	iconst_3		frem
	iconst_4		
	iconst_5	5	dadd
2	iadd		dsub
	isub		dmul
	imul		ddiv
	idiv		drem
	irem	6	ifeq
	iand		ifne
	ior		iflt
	ixor		ifle
3	ladd		ifgt
	lsub		ifge
	lmul	7	if_icmpeq
	ldiv		if_icmpne
	lrem		if_icmplt
	land		if_icmple
	lor		if_icmpgt
	lxor		if_icmpge

Table 2. Simple instruction subgroups

Subgroup	Instructions	Subgroup	Instructions
1	iload_0	2	istore_0
	iload_1		istore_1
	iload_2		istore_2
	iload_3		istore_3
3	lload_0	4	lstore_0
	lload_1		lstore_1
	lload_2		lstore_2
	lload_3		lstore_3
5	fload_0	6	fstore_0
	fload_1		fstore_1
	fload_2		fstore_2
	fload_3		fstore_3
7	dload_0	8	dstore_0
	dload_1		dstore_1
	dload_2		dstore_2
	dload_3		dstore_3

Table 3. Local storage related instruction subgroups

instructions like iload n, istore n etc. have two bytes length, while the instructions in Table 3 are one byte in length, substitution of the two different length instructions will affect all the following local variable's address, which leads to a more complicated processing, these instructions are also omitted.

We applied our scheme to five java programs (Table 4 and 5). It can be seen that about 10 percent of instructions are replaced by other instructions in the same group in Table 4, while only about 1 percent of instructions are replaced by other instructions in the same group in Table 5 for local storage related instructions. The file size in the second line of Table 4 gets smaller after obfuscated. This is due to the compress algorithm of jar compressed the obfuscated files more effectively. In fact, some of the class files inside the jar become bigger than the original.

Program	Size (bytes)		Total instructions	Substituted instructions
	Before obfuscation	After obfuscation		
Example.jar	111580	112480	2218	157
Imageviewer.jar	4137	4099	369	34
Jode.jar	530491	551630	83051	9909
Jbubblebreaker.jar	187795	189978	5718	649
JHEditor	77036	79942	11896	1545

Table 4. Simple instruction obfuscation

Program	Size (bytes)		Total instructions	Substituted instructions
	Before obfuscation	After obfuscation		
Example.jar	111580	112097	2218	25
Imageviewer.jar	4137	4196	369	11
Jode.jar	530491	565753	83051	2981
Jbubblebreaker.jar	187795	189086	5718	183
JHEditor	77036	79231	11896	567

Table 5. Local storage related instruction obfuscation

All the results in Fig. 13, 14, 15,16 are obtained by the following Mysaifu JVM settings:
Max heap size: 2M

- Java stack size: 32KB
- Native stack size: 160KB
- Verifier: Off

In debug mode, the max load delay is less than 10%, while most load delay is lower than 6% in release mode. When these programs are ready to run, their efficiency is the same as those original programs. Thus, our proposed scheme has little performance influence on the original program.

To some extent, this framework is a mixture of obfuscation, diversity, and tamper-proofing techniques. Each instruction is a one-to-many map to its semantics which is determined by the auxiliary input at runtime. Due to the fact that each output function λ_i can be defined independently based on different W_p, the translation space is very large. Only for Table 2, there will be 7! × 8! ×8! × 5! × 5! × 6! × 6! ≈ $5.1e10^{22}$ different rules (translators) approximately. Further more, suppose a program contains seventy instructions within

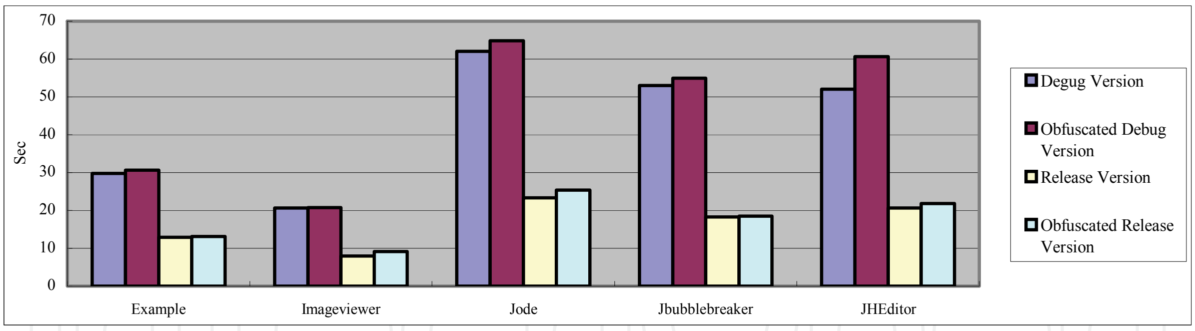


Fig. 13. Load time of simple instruction obfuscation in the Pocket PC 2003 SE Emulator

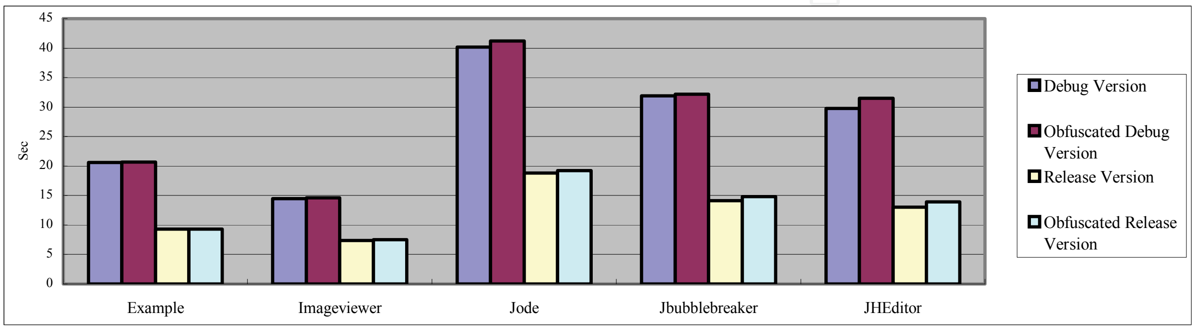


Fig. 14. Load time of simple instruction obfuscation in Dopod P800 mobile phone

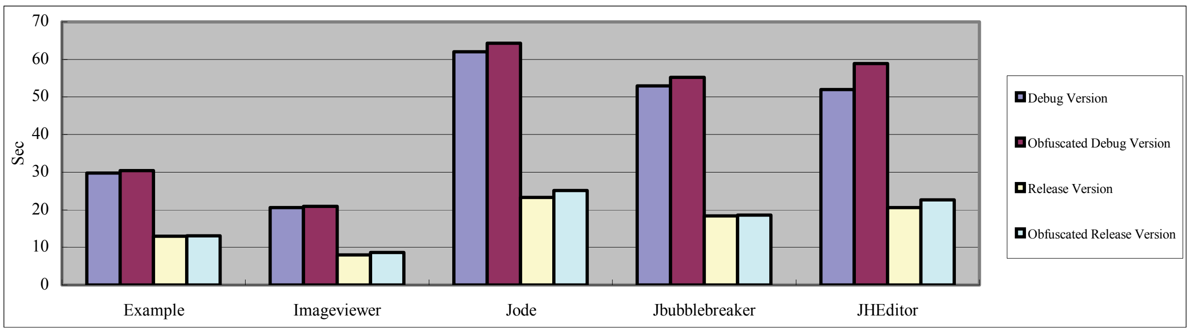


Fig. 15. Load time of local storage related instruction obfuscation in the Pocket PC 2003 SE Emulator

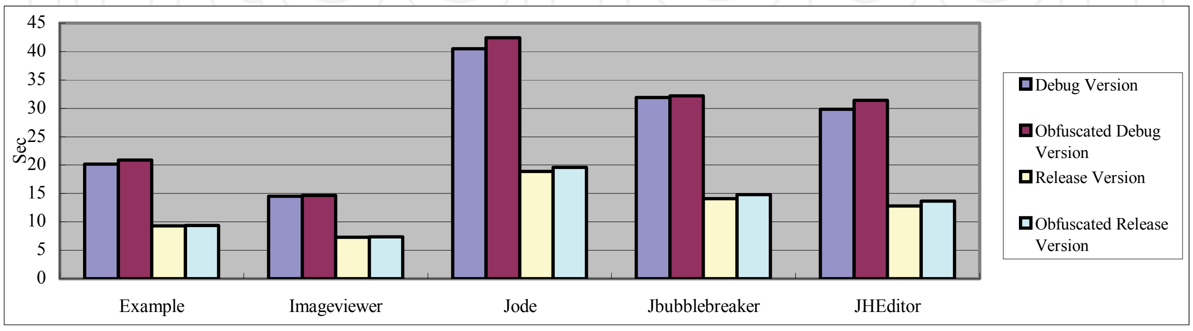


Fig. 16. Load time of local storage related instruction obfuscation in Dopod P800 mobile phone

which each ten instructions belongs to a distinct subgroup. Then there would be $10^7 \times 7 \times 8 \times 8 \times 5 \times 5 \times 6 \times 6 \approx 4$ e 10^{12} different versions even for a single interpreter. The translated program P_x can also be seen as an encrypted version. Not knowing the precise map among instructions, tampering it will definitely lead to the program's undefined behavior. An malicious users who tries to decompile the translated program only get the wrong semantics, he cannot reveal the real algorithm or design model.

The effective way to attack this framework is to crack Au_x. Once the SIM card number is obtained, the auxilary input Au_x can be easily decrypted. Using SIM card number as the enryption key is the most vulnerable weaknpoint of this model, it still need further study to establish a more secure way to protect Au_x.

6. Conclusion

Since Collberg's (1997) and Barak's (2001) seminal papers, program obfuscation has received considerable attentions over the last decade. As a result, a variety of formal definitions and practical methods of obfuscation have been developed. This chapter provides a brief survey of this progress on both the context of cryptography and software engineering. As a relatively less expensive method, despite the impossibility in cryptography, obfuscation does introduce the difficulty to reverse engineering. In this sense, it is still one of the promising techniques on software protection. In Sections 4, a call-flow obfuscation method is presented for Java program, which is also applicable to any other object oriented language. At the final section, an instruction obfuscation framework target at mobile phone Java applications is discussed. Different from personal computer platform, the JVM run in embedded system is usually customized according to different mobile phone hardware model, which leads to a large variety of JVMs. This kind diversity of JVM indicates that it is feasible and easy to apply the framework to the protection of mobile Java program.

7. References

Barak, B., Goldreich, O., Impagliazzo, R. ,Rudich, S., Sahai, A., Vadhan, S., Yang K. (2001). On the (Im)possibility of Obfuscating Programs. *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pp. 1-18, Santa Barbara, 2001 Aug. 19-23, Springer-Verlag, California.

Batchelder, M., Hendren, L. (2007). Obfuscating java: The most pain for the least gain, *Proceedings of the International Conference on Compiler Construction, Lecture Notes in Computer Science*, vol. 4420, 2007, pp. 96-110, Springer, Berlin.

Canetti, R. (1997). Towards Realizing Random Oracles: Hash Functions That Hide All Partial Information, *Advances in Cryptology - CRYPTO 1997, Lecture Notes in Computer Science*, vol. 1294, pp. 455–469, London, UK, 1997, Springer-Verlag.

Canetti, R., Micciancio, D., Reingold, O. (1998). Perfectly One-Way Probabilistic Hash Functions (Preliminary Version). *Proceedings of the 30th ACM Symposium on Theory of Computing (STOC 1998)*, pages 131–140. ACM Press.

Chan, J.T. , Yang, W. (2004). Advanced obfuscation techniques for Java bytecode, *Journal of Systems add Software*, Vol. 71, No.2, pp. 1~11.

Chatterjee, R., Ryder, B.G., Landi, W. (2001). Complexity of Points-To Analysis of Java in the Presence of Exceptions. *IEEE Transactions on Software Engineering*, Vol. 27, pp. 481-512.

Chow, S., Gu, Y., Johnson, H., Zakharov, V.A. (2001). An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs, *Proceedings of the 4th International Conference on Information Security*, pp. 144-155.

Cimato, S., Santis, A.D., Petrillo, U.F. (2005). Overcoming the obfuscation of Java program by identifier renaming, *Journal of Systems and Software*, Vol. 78, pp. 60-72.

Collberg, C., Thomborson, C., Low, D. (1997). A Taxonomy of Obfuscating Transformations. Tech. Report, #148, University of Auckland.

Collberg, C., Thomborson, C., Low, D. (1998a). Breaking Abstractions and Unstructuring Data Structures, *Proceedings of IEEE International Conference on Computer Languages*, pp. 28-38.

Collberg, C., Thomborson, C., Low, D. (1998b). Manufacturing cheap, resilient and stealthy opaque constructs. *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 184–196, San Diego, California, US.

Collberg, C., Thomborson, C. (2000). Watermarking, tamper-proofing, and obfuscation tools for software protection. Technical Report TR00-03, The Department of Computer Science, University of Arizona.

Cohen, F.B. (1993). Operating system protection through program evolution. *Computers and Security*, Vol. 12, pp. 565–584.

Diffie, W., Hellman, M. (1976). New directions in cryptography, *IEEE Transactions on Information Theory*, Vol. 22, No. 6, pp.644–654.

Drape, S. (2006). Generalising the array split obfuscation, *Information Sciences Journa*, Vol. 177, No. 1, pp. 202-219*l*, Elsevier, Department of Computer Science, University of Auckland, March 2006.

Drape, S., Majumdar, A., and Thomborson, C. (2007). Slicing Obfuscations: Design, Correctness, and Evaluation, *Proceedings of the Seventh ACM Workshop on Digital Rights Management (ACM-DRM 2007) (Part of the Fourteenth ACM Conference on Computer and Communications Security (ACM-CCS 2007))*, pp.70-81, October 29 - November 2, 2007. Alexandria, VA, USA. ACM Press.

Dyke, V., Colin, W. (2006). Advances in low-level software protection, PhD thesis, Oregon State University.

D'Anna, L., Matt, B., Reisse, A., Van Vleck, T., Schwab, S., LeBlanc, P. (2003). Self-Protecting Mobile Agents Obfuscation Report, Report #03-015, Network Associates Laboratories.

Ertaul, L., Venkatesh, S. (2005). Novel Obfuscation Algorithms for Software Security, *Proceedings of the 2005 International Conference on Software Engineering Research and Practice*, pp.209-215, June, Las Vegas.

Goldwasser, S., Kalai, Y.T. (2005). On the Impossibility of Obfuscation with Auxiliary Input. *Proceedings of the 46th Symposium on Foundations of Computer Science (FOCS 2005), IEEE Computer Society*, pp. 553–562, Washington, DC, USA.

Goldwasser, S., Rothblum, G. (2007). On Best-Possible Obfuscation, *Lecture Notes in Computer Science*, vol. 4392, pp.194–213,Springer-Verlag.

Hada, S.(2000). Zero-Knowledge and Code Obfuscation. In Tatsuaki Okamoto, editor, Advances in Cryptology - ASIACRYPT 2000, volume 1976 of Lecture Notes in Computer Science, pages 443–457, London, UK, 2000. Springer-Verlag.

Hofheinz, D., Malone-Lee, J., Stam, M. (2007). Obfuscation for Cryptographic Purposes. *Proceedings of 4th Theory of Cryptography Conference (TCC 2007), Lecture Notes in Computer Science,* vol. 4392, pp. 214–232. Springer-Verlag.

Hohenberger, S., Rothblum, G., Shelat, A., Vaikuntanathan, V. (2007). Securely Obfuscating Re-Encryption. *Proceedings of 4th Theory of Cryptography Conference (TCC 2007), Lecture Notes in Computer Science,* vol. 4392, pp. 233–252, Springer-Verlag.

Hohl, F. (1998) . Time limited blackbox security: protecting mobile agents from malicious hosts, *Mobile Agents and Security, Lecture Notes in Computer Science*, Vol. 1419, pp. 92–113. Springer, Heidelberg.

Josse, S. (2006). How to assess the effectiveness of your anti-virus. *Computer Virology*, 2006,Vol. 2, pp.51-65, Springer Paris.

Karnick, M., MacBride, J., McGinnis, S., Tang, Y., Ramachandran, R. (2006). A Qualitative analysis of Java Obfuscation, *Proceedings of 10th IASTED International Conference on Software Engineering and Applications*, Dallas TX,USA, November 13-15, 2006.

Kuzurin, N., Shokurov, A., Varnovsky, N., Zakharov, V. (2007). On the Concept of Software Obfuscation in Computer Security, *Lecture Notes in Computer Science,* Springer-Verlag, vol.4779, pp.281-298.

Linn, C., Debray, S. (2003). Obfuscation of executable code to improve resistance to static disassembly. *ACM Conference on Computer and Communications Security*, pp. 290-299, Washington D.C.

Lynn, B., Prabhakaran, M., Sahai, A. (2004). Positive Results and Techniques for Obfuscation. *Advances in Cryptology - EUROCRYPT 2004*, *Lecture Notes in Computer Science*, vol. 3027, pages 20–39. Springer-Verlag.

Monden, A., Monsifrot, A., Thomborson, C. (2004). A framework for obfuscated interpretation, *Australasian Information SecurityWorkshop (AISW2004)*, ed. P. Montague and C. Steketee, ACS, CRPIT, vol. 32, pp. 7–16.

Naccache, D., Shamir, A., Stern, J. P. (1999). How to copyright a function? , *Lecture Notes in Computer Science*, vol.1560, pp. 188–196, March 1999,Springer

Palsberg, J., Krishnaswamy, S., Kwon, M., Ma, D., Shao, Q., and Zhang, Y., (2000). Experience with software watermarking. *Proceedings of 16th IEEE Annual Computer Security Applications Conference (ACSAC'00)*. IEEE Press, p308-316. New Orleans, LA, USA, 2000.

Praveen, S., Lal, P.S. (2007). Array Data Transformation for Source Code Obfuscation. *Proceedings of World Academy of Science, Engineering and Technology (PWASET)* Vol.21 MAY 2007, ISSN 1307-6884.

Sosonkin, M., Naumovich, G., Memon, N. (2003). Obfuscation of design intent in object-oriented applications. *Proceedings of the 3rd ACM workshop on Digital rights management*, pp. 142–153, New York, NY, USA, 2003, ACM Press.

Toyofuku, T., Tabata, T., Sakurai, K. (2005). Program Obfuscation Scheme using Random Numbers to Complicate Control Flow, *Proceedings of the First International Workshop on Security in Ubiquitous Computing Systems (SecUbiq`05), Lecture Notes in Computer Science (LNCS)*, Vol. 3823, pp. 916-925.

Venkatraj, A. (2003). Program Obfuscation, MS Thesis, Department of Computer Science, University of Arizona.

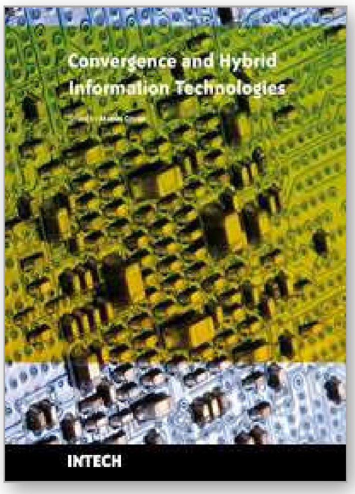
Wang, C., Hill, J., Knight, J.C., Davidson, J.W. (2001). Protection of software-based survivability mechanisms, *Proceedings of the 2001 conference on Dependable Systems and Networks, IEEE Computer Society*, pp. 193-202.

Wee, H. (2005). On Obfuscating Point Functions, *Proceedings of the 37th ACM Symposium on Theory of Computing (STOC 2005)*, pp. 523–532, New York, NY, USA, 2005. ACM Press.

Wyseur, B. (2009). White-Box Cryptography, PhD thesis, Katholieke Universiteit Leuven.

Wroblewski, G. (2002). General Method of Program Code Obfuscation, PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics.

Zhu, F. (2007). Concepts and Techniques in Software Watermarking and Obfuscation, PhD Thesis, University of Auckland.



ISBN 978-953-307-068-1
Hard cover, 426 pages
Publisher InTech
Published online 01, March, 2010
Published in print edition March, 2010

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821