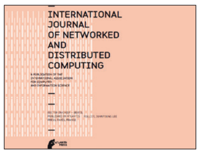
International Journal of Networked and Distributed Computing
Vol. **6**(**4**); September (2018), *pp.* 232–238
DOI: 10.2991/ijndc.2018.6.4.6; ISSN 2211-7946
https://www.atlantis-press.com/journals/ijndc



Design and Evaluation of the De-obfuscation Method against the Identifier Renaming Methods

Yosuke Isobe[1,*], Haruaki Tamada[2]

[1]*Division of Frontier Informatics, Graduate School of Kyoto Sangyo University, Motoyama, Kamigamo, Kita-ku, Kyoto, Kyoto 603-8555, Japan*
[2]*Faculty of Information Science and Engineering, Kyoto Sangyo University, Motoyama, Kamigamo, Kita-ku, Kyoto, Kyoto 603-8555, Japan*

ARTICLE INFO

Article History
Received 27 June 2018
Accepted 24 September 2018

Keywords
De-obfuscation
identifier renaming method
software protection
random forest

ABSTRACT

The Identifier Renaming Method (IRM) is a well-used obfuscation method since almost obfuscation tools use the algorithm, and easy to implement. The IRM transforms the identifier names in the programs to meaningless names in order to hard to understand. However, the evaluations against attacks such as de-obfuscation were not conducted. Therefore, this paper proposes the method for restoring the verbs in method names from identifier renamed programs. This approach is one of the attack of the de-obfuscation, thus, this paper evaluates IRM tolerance against the attack. From the experimental evaluation, the proposed method can restore the 49.71% of method names to the original verbs. Furthermore, focusing on the meanings of verbs, the proposed method recommends the verbs of similar meanings to the original verbs in 57.01% of methods.

1. INTRODUCTION

Unfortunately, some cracking incidents reported. For example, Nintendo Switch Online is the online service for playing the games provided with it and released on September 18, 2018[1]. However, Kuchera and Frank [1] reported that Nintendo Switch Online was already cracked, and own ROMs can be played using the cracked Nintendo Switch.

To protect the software from cracking, the obfuscation methods were proposed. An obfuscation method is to change programs hard to understand by preserving input/output specification. Various obfuscation methods are proposed [2–4], and are implemented to tools, such as Dash-O[2], ProGuard[3], and etc. In particular, the Identifier Renaming Method (IRM) is widely used, which changes the identifiers in the programs to meaningless names. Because it is easy to implement and to understand the approach [5,6]. However, IRM did not sufficiently discuss the tolerance against attacks, and strength of the protection. Because the relationships between the meanings of identifiers and strength of the protection are not clear.

Now, we consider a scenario that an adversary steals programs and obfuscates them by an IRM. The original authors of the programs try to analyze the programs by some methods, such as decompilation. The analysis of the programs is quite hard since an IRM eliminates the meanings of identifiers. If a technique gives meanings to methods of programs, it is useful for the analysis. Of course, it is a security risk that adversaries conduct the technique. However, the obfuscation method should tolerate some attacks and should be evaluated the tolerance. Therefore, this paper proposes the de-obfuscation method for the identifier renamed programs and evaluates the tolerance of IRM. If we succeed in de-obfuscation by the proposed method, it can be said that tolerance of IRM is low.

For this, we try to restore identifiers from programs applying IRM, illustrated in Fig. 1. Usually, programs have several identifiers, such as class, method, and variable names. We focus on, especially, method names in programs. Because the clues for restoring the variable names is few since they strongly depend on the domain of programs. Also, class names are hard to restore, too, since they depend on variable names. Therefore, at first, the paper proposes to restore method names in programs from their opcode list. Especially, this paper focuses on restoring verbs in method names. Because some verbs in method names relate to their behaviors. Thus, the goal of this paper is to recommend the verbs of the method names.

For restoration, we use the random forest which is one of the machine learning algorithms. The proposed method, at first, constructs a restoration model from a huge number of programs. Then, the model recommends the names from an opcode list of methods. The opcode list is independent variable (explanatory variable), and the response variable (objective variable) is a verb of method names.

The remainder of the paper first defines the software obfuscation method and IRM (Section 2). Next, proposes a method for restoring the method names (Section 3). Afterward, we conduct experiments to evaluate the proposed method (Section 4), then discuss

* *Corresponding author. Email: i1788016@cc.kyoto-su.ac.jp*

[1] https://www.nintendo.com/switch/online-service/
[2] https://www.preemptive.com/products/dasho/
[3] https://www.guardsquare.com/en/proguard

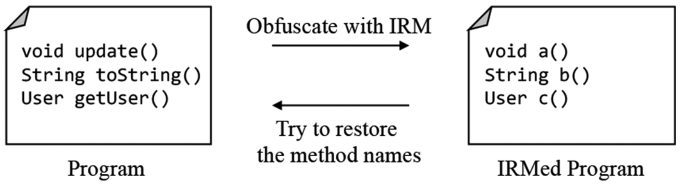


Figure 1 | The purpose of the proposed method

the threats to validity (Section 5). Finally, the conclusion will be shown (Section 7).

2. PRELIMINARY

2.1. Program Obfuscation Method

This section formulates the notion of program obfuscation method [7,8]. We start with the definition of the program understanding since the obfuscation prevents malicious users from understanding the program.

Definition 1. (Program understanding cost): Let p be a given program and X be a set of information included in p. When a user can extract X from p by a certain method, then we define that the user has understood p about X. For this, we denote a cost of the understanding as $c(p, X)$ in an abstract manner. The cost would be characterized by, for example, the time, efforts, the necessary knowledge, devices, etc., taken for the analysis. Then, we give a general definition of the program obfuscation.

Definition 2. (Program obfuscation): Let p be a given program, X be a given a set of information of p, I be an input set of p, and $r(p, I)$ be an output set of p with I. Then, the obfuscation of p with respect to X is to translate p into p' with a certain method T (i.e., $p' = T(p)$), such that

Condition 1: $r(p, I) = r(p', I)$

Condition 2: $c(p, X) < c(p', X)$.

Condition 1 means input/output mapping of the program are preserved before and after obfuscation. The obfuscation must preserve the external specification of the target program. Also, Condition 2 shows that extracting X from p' is more difficult than p.

2.2. Identifier Renaming Method

An identifier renaming method is one of the obfuscation methods. The identifier renaming method is widely used since almost obfuscation tools use the method. The method replaces each name in the program with another, to hide information reasoned from the name. Note that the replacing provides no effects for the program execution. Because names in a program are just identifiers for the computers.

Definition 3. [Identifier Renaming Method (IRM)]: Let p be a given program, U_p be a set of all name appeared in p, and $N_p(\subset U_p)$ be a set of names, which are targeted on the obfuscation. An identifier renaming method for p is to replace each name $n \in N_p$ in p to other name $n'(= t(n))$, and to obtain an obfuscated program p', where t is one-to-one mapping ($t : N_p \rightarrow N_{p'}(N_{p'} \subset U_{p'})$).

If p is an object-oriented program, a name appears in a class, a method, a field or a local variable. The names are shown in the declaration and the reference parts. The IRM changes the names appeared in the declaration part, therefore, involving the reference part.

3. THE PROPOSED METHOD

3.1. Key Idea

The program generally uses a lot of names such as class, method, and field. The names appearing in the program are categorized into the declaration and the reference parts. In the declaration part, the name is shown in the method and the variable names. In the reference part, the name is used as calling the methods, and assigning and loading the variable. Almost IRMs change the names appeared in the declaration and the reference parts are changed as a side effect. For this, this paper focuses on restoring (de-obfuscating) names of methods in declaration parts. Note that the IRM targeted on reference part does not support in this paper [8].

The de-obfuscation needs to acquire the clues from the obfuscated program. The remained clues are opcodes in the methods because IRM does not affect them. However, the IRM eliminates the meanings of names contained in the original programs. Moreover, another obfuscation method might be applied for opcodes. Therefore, it is impossible to completely de-obfuscate the IRM, since the clues are not enough.

On the other hand, there are many software repositories in the world, today. The repositories have a huge number of libraries, and programs. This paper tries to recommend the original or similar names by using the opcodes of methods from the programs in the repositories.

However, it is still difficult to restore the method names. The method names are usually composed of a verb and an object (e.g., isEmpty and getBytes). The object depends on the field variable, class names or etc. In IRM, the meanings of all names are eliminated. Therefore, it is a quite hard task to guess the names of the objects. In this paper, we focus on restoring the verb part of the method names. Figure 2 illustrates the key idea of the proposed method. From Fig. 2, the response variable is the verb of the method name, and the clues (independent variable) is the list of opcodes.

Note that, the target of our method is the Java language. However, the proposed method can port to other platforms, easily.

3.2. Procedures of the Proposed Method

3.2.1. Overview

The goal of this paper is to recommend verbs for the method name from opcodes. For this, we construct the restoration model from opcodes of a huge number programs in the software repositories. The model recommends verbs of each method, therefore, the obfuscated program by IRM is de-obfuscated by applying the model.

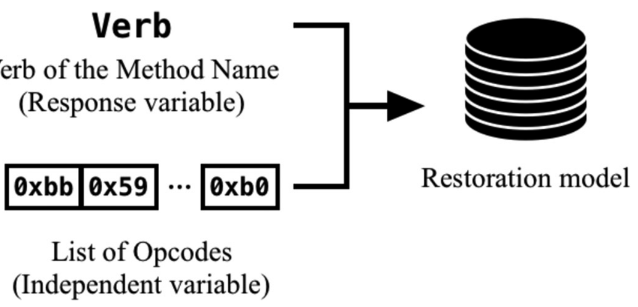


Figure 2 | The key idea of the proposed method

The procedure of the proposed method consists of the following four phases.

1. Collecting the learning data.
2. Converting the collected data for the model.
3. Constructing the restoration model.
4. De-obfuscating the target programs.

3.2.2. Collecting the learning data

At phase 1, at first, software repositories are chosen for collecting a program set. Then, a program set $P = \{p_1, p_2, ..., p_n\}$ is constructed from selected repositories. The collected programs are adjusted to the learning data format to construct the restoration model. Next, we extract a method information set $M_i = \{m_{i,1}, m_{i,2}, ..., m_{i,l}\}$ from each $p_i (1 \le i \le n)$. $m_{i,j} (1 \le i \le n, 1 \le j \le l)$ has two properties, a method name $c_{i,j}$ and an opcode list $O_{i,j} = \{b_1^{i,j}, b_2^{i,j}, ..., b_z^{i,j}\}$ $(m_{i,j} = \{c_{i,j}, O_{i,j}\})$.

3.2.3. Converting the collected data for the model

In the phase, each method information $m_{i,j}$ is transformed to $m'_{i,j}$ by the following four steps for the input of the machine learning. The first step in the phase is to extract the first verb from $c_{i,j}$ and obtain $t_{i,j}$. For example, the method name getSize is transformed into get. Some methods do not start with verbs, we eliminate them from M_i.

In the second step, we normalize each $t_{i,j}$ to $\tau_{i,j}$. This normalization converts with the following rules.

- A verb for third person singular form into base form (e.g., equals to equal).
- A verb in abbreviation form to non-abbreviation form (e.g., auth, init, and calc to authenticate, initialize, and calculate).
- Inconsistent spelling into unified form (e.g., analyze, and analyse to analyze).

Note that, we have considered seven words new, setup, cleanup, init, calc, to, and as as verbs since these words are often used as words similar to verbs, based on previous studies [8].

In the third step, each $b_k^{i,j} \in O_{i,j}$ is merged by its meanings. For example, several store opcodes exist in Java virtual machine (e.g. istore, lstore, and etc.) [9]. However, the meaning of each store opcode is same, which pops a value from the stack and stores it into the local variable. The differences of the opcodes of same meanings are the targeted types, istore is for int type, and lstore is for long type. Therefore, $b_k^{i,j}$ is converted into $o_k^{i,j}$ by merging with the rules shown in Table 1. The resultant opcode list is $O_{i,j} = \{o_1^{i,j}, o_2^{i,j}, ..., o_z^{i,j}\}$, and the resultant data are $m'_{i,j} = \{\tau_{i,j}, O_{i,j}\}$.

In the final step in the phase, we vectorize the opcode list $O_{i,j}$ to $\mathcal{V}_{i,j} = \{\{o_1^{i,j}, a_1^{i,j}\}, ..., \{o_z^{i,j}, a_z^{i,j}\}\}$ by the number of appearances. $a_k^{i,j}$ is the number of appearing $o_k^{i,j}$ in $O_{i,j} (1 \le k \le z)$. Finally, $f_{i,j} = \{\tau_{i,j}, \mathcal{V}_{i,j}\} (1 \le i \le n, 1 \le j \le l)$ is obtained which converted from $m'_{i,j}$.

3.2.4. Constructing the restoration model

In this phase, we construct a restoration model from $f_{i,j}$ shown in Section 3.2.3. The random forest is selected for the machine learning algorithms to construct the restoration model. The objective variables are $\tau_{i,j}$ and the explanatory variables are $O_{i,j}$.

Table 1 | The merge rules for JVM instructions

Group	Opcode
STACK	nop, pop, pop2, dup, dup_x1, dup_x2, dup2, dup2_x1, dup2_x2, swap
CONSTANT	aconst_null, iconst_X, lconst_X, fconst_X, dconst_X, bipush, sipush, ldc, ldc_w, ldc2_w
LOAD	iload, lload, fload, dload, aload, iload_X, lload_X, fload_X, dload_X, aload_X
ARRAY	iaload, laload, faload, daload, aaload, baload, caload, saload, iastore, lastore, fastore, dastore, aastore, bastore, castore, sastore, arraylength, multianewarray
STORE	istore, lstore, fstore, dstore, astore, istore_X, lstore_X, fstore_X, dstore_X, astore_X
ADD	iadd, ladd, fadd, dadd, iinc
SUBTRACT	isub, lsub, fsub, dsub
MULTIPLY	imul, lmul, fmul, dmul
DIVIDE	idiv, ldiv, fdiv, ddiv
REMAIN	irem, lrem, frem, drem
NEGATE	ineg, lneg, fneg, dneg
SHIFT_LEFT	ishl, lshl
SHIFT_RIGHT	ishr, lshr
USHIFT_RIGHT	iushr, lushr
AND	iand, land
OR	ior, lor
XOR	ixor, lxor
CAST	i2l, i2f, i2d, l2i, l2f, l2d, f2i, f2l, f2d, d2i, d2l, d2f, i2b, i2c, i2s, checkcast
COMPARE	lcmp, fcmpl, fcmpg, dcmpl, dcmpg, instanceof
BRANCH	ifeq, ifne, iflt, ifge, ifgt, ifle, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, if_acmpeq, if_acmpne, goto, jsr, tableswitch, lookupswitch, ifnull, ifnonnull, goto_w, jsr_w
RETURN	ireturn, lreturn, freturn, dreturn, areturn, return, ret
FIELD	getstatic, putstatic, getfield, putfield
INVOKE	invokevirtual, invokespecial, invokestatic, invokeinterface, invokedynamic
NEW	new, newarray, anewarray
THROW	athrow
OTHERS	monitorenter, monitorexit, wide

Therefore, the model outputs verbs of the methods from their opcode list.

3.2.5. De-obfuscating the target program

This phase de-obfuscates the given obfuscated programs by IRM by the restoration model in Section 3.2.4. The inputs for the model is the vectorized opcode list from the given obfuscated programs. Therefore, we extract vectorized opcode list $O_{i,j}$ from given programs in the same way as in Section 3.2.2. Finally, the model recommends a candidate verb v for each method from the vectorized opcode list.

4. EXPERIMENTAL EVALUATION

4.1. Research Questions

We evaluate the proposed method through the following three research questions.

- **RQ1:** How much can the proposed method restore the original verb?
- **RQ2:** Is the sequence of opcodes important?
- **RQ3:** Is the proposed method useful?

4.2. Experimental SetUp

4.2.1. The restoration model

For the experiment, we obtain programs as learning data from the Maven central repository (MCR) [10]. Then, the program set P was obtained from MCR and obtained 17,714 jar files which are the newest version of each product. Next, we extracted 21,738,029 methods from all classes in the jar files. The constructors (<init>) and static initializers (<clinit>) are eliminated from the extracted methods. Because the names of them are fixed in JVM and the Java languages.

We applied phase 2 shown in Section 3.2.3 to the extracted methods. Small and too large methods are eliminated. Because those methods are too few or too much information and are noise for the restoration. It is [30, 1000] that the range of opcodes length for the target methods. This range is about 10–300 lines of code.

Finally, we obtained 2,404,277 methods and 1,813 verbs. However, the usage frequencies of the verbs vary widely. Therefore, the experiments narrow down to the top 20 of usage frequencies, and resultant methods are 935,796 methods. The verbs in the top 20 of usage frequencies were denoted $T = \{\tau_1,\ldots,\tau_{20}\}$. Before choosing T, we eliminate verbs get and set, because clues of them for restoration are usually little, and resultant verbs are not very useful.

4.2.2. Test data

The test data are shown in Table 2. Those programs were randomly chosen from sonartype releases maven repository[4]. That is, the restoration model does not include the programs for test data. Moreover, the method information was extracted from those programs, and we narrow the target methods with the method name in T.

Besides, we developed python script to conduct above procedures with scikit-learn[5] on Python 2 platform. Also, the experiments were conducted on macOS High Sierra (10.13.3), MacBook Pro, 2.7 GHz Intel Core i5, 16 g.

Table 2 | The programs for the test data

Name	Data size
BTSync-Java-0.1.jar	5.2 MB
DynamicJasper-core-fonts-1.0.jar	2.9 MB
acm-2.0.0-preview-5.jar	251 KB
amqp-scala-client_2.12-2.0.0.jar	555 KB
api-doc-0.0.34.jar	554 KB
bitcoinj-core-0.15-cm04.jar	1.5 MB
codedeploy-notifications_2.11-0.2.1.jar	208 KB
dw-jdbc-0.4.jar	167 KB
elasticsearch-5.0.0-beta1.jar	8.9 MB
flink-kudu-connector-1.0.jar	34 MB
geopackage-core-1.3.1.jar	314 KB
itk-payloads-0.5.jar	1.8 MB
ixa-pipe-chunk-1.1.0.jar	5.3 MB
ixa-pipe-parse-1.1.1.jar	59 MB
mlapi_2.12-0.0.1.jar	367 KB
monetdb-java-lite-2.33.jar	6.4 MB
no-exceptions_2.11-1.0.1.jar	173 KB
openfin-desktop-java-adapter-6.0.1.0.jar	310 KB
orbit-runtime-1.1.0.jar	1.9 MB
payara-microprofile-1.0-4.1.2.172.jar	37 MB
phtree-0.3.1.jar	336 KB
scala-expect_2.12-6.0.0.jar	199 KB
scenery-0.2.2.jar	1.5 MB
schemaspy-maven-plugin-1.2.1.jar	280 KB
semanticvectors-5.8.jar	12 MB
siren-join-2.4.5.jar	210 KB
uaiMockServer-1.2.5.jar	579 KB
vldocking-3.0.4.jar	392 KB

4.3. How Much can the Proposed Method Restore the Original Verb?

This RQ investigates the success rate for the restoration. The experimental overview is shown in Fig. 3. The procedure of the experiment is that we (1) construct the restoration model from the learning data shown in Section 4.2.1, (2) restore the names for test data (Section 4.2.2), and (3) evaluate the names restored from the model. In the (3), the evaluation examines that the names restored from the model and original names are the same.

The result of the experiment shows that the success rate for restoration was 39.19% (5,740/14,647). In other words, 39.19% of verbs are restored to original verbs. The confusion matrix for each verb is shown in Table 3. The columns of TP, FP, FN, and TN show the count in true positive, false positive, false negative and true negative, respectively. Also, the columns of P and R indicate precision and recall. From Table 3, the precision (P) and recall (R) vary widely

[4] https://oss.sonatype.org/content/repositories/releases/

[5] http://scikitlearn.org/stable/

equal and to were relatively high, precision and recall exceed 75% and 55%, respectively. This result was considered by two reasons. The first reason is equals and toString methods are defined in java.lang.Object which is the root class of the inheritance hierarchy in the Java language. The second reason is the process of almost methods of equals and toString are quite similar.

On the other hand, the precision and recall of do and test are low, under 20% and 15%, respectively. Therefore, it was hard to recommend the verb by the proposed method, since there is no typical process in the methods of those names.

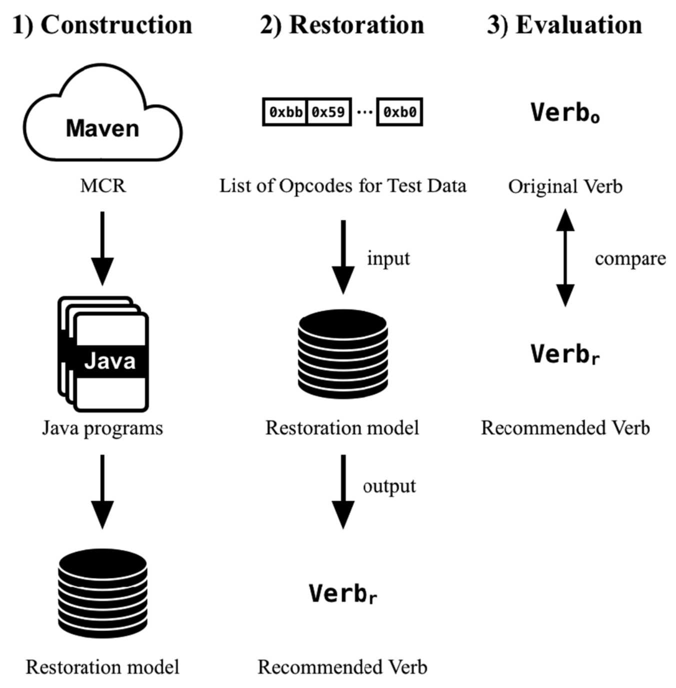


Figure 3 Overview of the experimental evaluation

Table 3 Restoration results: confusion matrix

	TP	FP	FN	TN	*P* (%)	*R* (%)
add	442	1667	459	12079	21.0	49.1
apply	152	210	462	13823	42.0	24.8
be	520	950	170	13007	35.4	75.4
check	130	352	342	13823	27.0	27.5
copy	58	105	223	14261	35.6	20.6
create	453	1599	595	12000	22.1	43.2
do	77	492	484	13594	13.5	13.7
equal	946	86	397	13218	91.7	70.4
find	159	633	254	13601	20.1	38.5
generate	46	133	110	14358	25.7	29.5
initialize	64	218	259	14106	22.7	19.8
parse	238	363	559	13487	39.6	29.9
process	55	209	247	14136	20.8	18.2
read	428	733	602	12884	36.9	41.6
remove	242	361	291	13753	40.1	45.4
run	118	179	1254	13096	39.7	8.6
test	10	40	404	14193	20.0	2.4
to	1032	301	773	12541	77.4	57.2
visit	187	85	307	14068	68.8	37.9
write	383	191	715	13358	66.7	34.9

From the result, the answer to RQ1 was that the proposed method successfully restored in 40% verbs. However, the precision of the restoration strongly depends on verbs. The restoring success rate is high in the methods have typical processes. The verbs which have various meanings are hard to restore to the original verbs since the processes in the methods are various.

4.4. Is the Sequence of Opcodes Important?

In RQ1, the restoration model was constructed from the appearance frequencies of the opcodes by vectorization. In addition, in the RQ2, we also focus the order of opcodes to use 2 g of the opcodes. The 2 g is the two contiguous opcodes and shows the relationships between them.

In RQ2, the new restoration model was constructed from the appearance frequencies of the 2 g by vectorization. We conducted the experiment by the new model from the learning data shown in Section 4.2.1, then applied the model to test data shown in Table 2.

In the result of RQ2, the success rate for restoration was 49.71% (7,281/14,647), the result was improved about 10% from the result of RQ1. Also, the confusion matrix in the experiment is shown in Table 4. The columns of Table 4 are the same as Table 3. Table 4 shows that precisions and recalls are improved from Table 3 in all verbs except precision of write.

From the result, the restoration model of the RQ2 is useful for restoring the verbs. Therefore, the answer of RQ2 was yes, it is important not only the appearance frequencies but also the relationships between the opcodes.

4.5. Is the Proposed Method Useful?

The final RQ examines whether the proposed method provides useful information for analyzing the identifier renamed programs.

Table 4 Restoration results: confusion matrix (2 g)

	TP	FP	FN	TN	*P* (%)	*R* (%)
add	529	1632	372	12114	24.5	58.7
apply	283	178	331	13855	61.4	46.1
be	573	440	117	13517	56.6	83.0
check	160	256	312	13919	38.5	33.9
copy	91	71	190	14295	56.2	32.4
create	543	1311	505	12288	29.3	51.8
do	77	385	484	13701	16.7	13.7
equal	1221	52	122	13252	95.9	90.9
find	187	382	226	13852	32.9	45.3
generate	66	127	90	14364	34.2	42.3
initialize	125	321	198	14003	28.0	38.7
parse	277	276	520	13574	50.1	34.8
process	74	237	228	14108	23.8	24.5
read	480	290	550	13327	62.3	46.6
remove	276	233	257	13881	54.2	51.8
run	173	183	1199	13092	48.6	12.6
test	85	218	329	14015	28.1	20.5
to	1141	368	664	12474	75.6	63.2
visit	290	70	204	14083	80.6	58.7
write	630	336	468	13213	65.2	57.4

The RQ1 and 2 investigated that can the proposed method restore the original verbs from the opcodes. However, similar verbs with the originals would be also useful, therefore, it was not necessary to the correct restoration. For example, to and convert, and perform and run are quite similar, therefore, either verb should be correct restoration result. Hence, the experiment calculates the similarities between original and recommended verbs using WordNet[6]. We used path similarity which calculates the reciprocal of the length of the path through the common hypernym [11]. Since, the WordNet cannot calculate the similarity between different parts of speeches, we used almost synonymous verbs for the words shown in Section 3.2.2. The experiment used the restoration model constructed in RQ2. Moreover, test data were all methods of programs shown in Table 2, 37,395 methods.

The result of the experiment shows that the success rates were 21.81%, 29.84%, and 57.01% when the path similarities were 1 (synonym), 1/2 (hypernym), and 1/3, respectively. The success rate from the results was lower than RQ1 and 2 since the number of test data increased [14,627 (RQ1) to 37,395 (RQ3) methods, about 2.5 times]. Therefore, the answer of RQ3 was yes, the proposed method can provide verbs with similar meanings to the originals.

5. THREATS TO VALIDITY

5.1. The Learning Data and Test Data

In the experiments, the learning and test data were obtained from Maven repositories. Those data were selected automatically, however, the results of the experiments depends on them. Moreover, the suitabilities of original names were not evaluated. It means that the names in the learning and test data were potentially not suitable, e.g., names and contents of methods are unmatched, or no meanings in the names. The unsuitable names in the learning data will derive the wrong restoration. However, if the number of unsuitable names was few, the impact to the results are little. On the other hand, unsuitable names in the test data affect the results.

5.2. The Contribution to De-obfuscation

The goal of this paper is to restore the verb in the method names, and this paper showed the restoring methods of the verb from method contents. However, we did not evaluate the contribution to the tolerance of IRM by our restoration method. In other words, we have to evaluate the effect of the de-obfuscation against IRM by our method in our future work.

5.3. Inconsistency against Instinct

In the experiment for RQ3, the similarities between words were calculated using WordNet. The similarities are calculated based on not the programming language, but natural languages. The meanings in the programming and the natural languages generally have gaps. Especially, in the case of path similarity was 1/3, we did not evaluate whether the results from the proposed method were certainly suitable.

Table 5 shows example pairs of the original and recommended names. The names in first two columns are relatively similar meanings in use of programs, therefore, the proposed method succeeded in the restoration. However, the meanings of last two rows are quite different. In the case of last two rows, the proposed method failed to restore. Some results are similarly suitable to original names, and other results are the wrong restoration. Thus, in our future works, we will create a dictionary for programming languages.

Table 5 | Examples of original and recommended name pair

Original	Recommended
start	run
translate	convert
read	write
accept	remove

5.4. Clues for the Restoration

This paper used only the opcodes of the methods as the clues for restoration. Since IRM keeps types defined in the system libraries, the types might be the clues for restoration. Other information in the programs which keeps by the IRM would improve the success rate of the restoration. However, other obfuscation techniques might modify them. It is our future works to improve the success rate by using other clues.

5.5. Restoration Target Verbs

This paper targeted top 20 verbs in the learning data for restoration. In our previous works, the restoration model was constructed by using all verbs [12]. The resultant success rate was 6.4, quite low. However, investigating frequencies of the verbs were 22 types of verbs occupy 90% of the recommended verbs. The 22 types of verbs were shown in Table 3, and get and set. Even if using all verbs for the model did not improve the recommendation results. Also, in the test data, 20 verbs in *T* occupy the 40% of verbs. Therefore, narrowing targets are good practice for de-obfuscating IRM.

6. RELATED WORKS

Currently, there are a few researchers to study the evaluation of the obfuscation methods. For example, Udupa et al. [13] proposed the de-obfuscation technique of the basic block flattering method (one of control flow obfuscations) using the static and dynamic analyses. In data flow obfuscations, it is evaluated itself by defining metrics. Kanzaki et al. [14] evaluated the obfuscation methods by to measure program stealthiness from the artificiality of opcodes. Also, Cimato et al. [15] proposed the de-obfuscation method of IRM. However, the restoring targets of Cimato et al. were variable names, not method names.

On the other hand, the analysis methods focused on the names were proposed. Kashiwabara et al. [6] proposed the recommended method for verbs in method names. The clues of recommendation are class and method names used in a method. However, their major targets are Java source codes, therefore, the method is not used in de-obfuscation.

[6] https://wordnet.princeton.edu/

7. CONCLUSION

This paper evaluated the tolerance of IRM, and restored the verbs of methods by the random forest. For the clues of the restoration, we focus on the opcodes in the methods. The experimental evaluations show that the success rate for the restoration was 49.71% by using the appearance frequencies of opcodes' 2 g as clues. Also, experiment focuses on meanings of verbs was conducted and shows that the success rate was 57.01% when the path similarities in WordNet was 1/3. In our future works, we improve the success rate by narrowing targets, using other clues, and updating algorithms for calculating similarities between verbs.

ACKNOWLEDGMENTS

Part of this work was supported by JSPS KAKENHI Grant Numbers 17K00196, 17K00500, and 17H00731.

REFERENCES

[1] B. Kuchera, A. Frank, Nintendo Switch Online's NES emulator already hacked to allow more games, https://www.polygon.com/2018/9/19/17879042/nintendo-switch-onlines-nes-emulator-hacked, 2018 (Last accessed 7 October 2018).

[2] C. Collberg, J. Nagra, Surreptitious Software: Obfuscation, Watermarking, and Tamper-proofing for Software Protection, Addison-Wesley, Boston, USA, 2009.

[3] K. Fukuda. H. Tamada, To prevent reverse-engineering tools by shuffling the stack status with hook mechanism, Int. J. Software Innov. 3 (2015), 14–25.

[4] S. Qing, W. Zhi-yue, W. Wei-min, L. Jing-liang, H. Zhi-wei, Technique of source code obfuscation based on data flow and control flow transformations, in: Proceedings of the 7th International Conference on Computer Science & Education (ICCSE 2012), Melbourne, Australia, July 2012.

[5] E.W. Høst, B.M. Østvold, The Programmer's Lexicon, Volume I: the verbs, in: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), Paris, France, 2007, pp. 193–202.

[6] Y. Kashiwabara, Y. Onizuka, T. Ishio, Y. Hayase, T. Yamamoto, K. Inoue, Recommending verbs for rename method using association rule mining, in: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 3–6 Feburary 2014, pp. 323–327.

[7] H. Tamada, M. Nakamura, A. Monden, K-i. Matsumoto, Introducing dynamic name resolution mechanism for obfuscating system-defined names in programs, in: Proceedings of the IASTED International Conference on Software Engineering (IASTED SE 2008), Innsbruck, 12–14 February 2008, pp. 125–130.

[8] Y. Kashiwabara, T. Ishio, H. Hata, K. Inoue, Method verb recommendation using association rule mining in a set of existing projects, IEICE Trans. Inf. Syst. 98 (2015), 627–636.

[9] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, The Java Virtual Machine Specification. Oracle, Inc., Java SE 7 edition, 2013. https://docs.oracle.com/javase/specs/jvms/se7/html/index.html.

[10] S. Raemaekers, A. van Deursen, J. Visser, The Maven repository dataset of metrics, changes, and dependencies, in: Proceedings of the 2013 10th IEEE Working Conference on Mining Software Repositories (MSR 2013), IEEE, San Francisco, CA, USA, 18–19 May 2013, pp. 221–224.

[11] T. Pedersen, S. Patwardhan, J. Michelizzi, WordNet::Similarity: measuring the relatedness of concepts, in: Proceedings of the HLT-NAACL-Demonstrations 2004, pp. 38–41.

[12] Y. Isobe, H. Tamada, De-obfuscating identifier renaming methods by random forest, in: Proceedings of the 24th workshop of Foundation of software engineering (FOSE 2017), November 2017, pp. 93–98 (In Japanese).

[13] S.K. Udupa, S.K. Debray, M. Madou, Deobfuscation: reverse engineering obfuscated code, in: Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005), IEEE, Pittsburgh, PA, USA, 2005.

[14] Y. Kanzaki, A. Monden, C. Collberg, Code artificiality: a metric for the code stealth based on an n-gram model, 2015 IEEE/ACM 1st International Workshop on Software Protection (SPRO 2015), IEEE, Florence, 2015, pp. 31–37.

[15] S. Cimato, A. de Santis, U. Ferraro Petrillo, Overcoming the obfuscation of Java programs by identifier renaming, J. Syst. Software. 78 (2005), 60–72.