Code Artificiality: A Metric for the Code Stealth Based on an N-gram Model

Yuichiro Kanzaki
National Institute of Technology
Kumamoto College
Koshi, Kumamoto, Japan
kanzaki@kumamoto-nct.ac.jp

Akito Monden
Nara Institute of Science and Technology
Ikoma, Nara, Japan
akito-m@is.naist.jp

Christian Collberg
University of Arizona
Tucson, Arizona, USA
collberg@gmail.com

***Abstract*—This paper proposes a method for evaluating the artificiality of protected code by means of an N-gram model. The proposed artificiality metric helps us measure the stealth of the protected code, that is, the degree to which protected code can be distinguished from unprotected code. In a case study, we use the proposed method to evaluate the artificiality of programs that are transformed by well-known obfuscation techniques. The results show that static obfuscating transformations (e.g., control flow flattening) have little effect on artificiality. However, dynamic obfuscating transformations (e.g., code encryption), or a technique that inserts junk code fragments into the program, tend to increase the artificiality, which may have a significant impact on the stealth of the code.**

I. Introduction

To date, attacks by an end user who attempts to obtain security-sensitive information in software products, called man-at-the-end (MATE) attacks, have been creating a serious threat to software industry. In order to protect software against such attacks, various methods for protecting software, such as program obfuscation and program encryption, have been proposed [1].

Assuming that the adversary goes through a typical *locate-alter-test* cycle [1], the *stealth* of the protected code, that is, the degree to which protected code can be distinguished from unprotected code, has a large effect on the strength of the protection. For instance, let us consider a scenario in which a conditional branch of software needs to be protected against adversaries, and the part of a binary machine code is encrypted. In such case, adversaries can easily determine that this encrypted part is not in its original form, because the code will now include many unusual instructions. Such unstealthy code can expose the protected (secret) part of the program, giving adversaries a significant clue to its location. As a result, the protected code can drastically reduce the effort that adversaries will have to invest for the attack.

In spite of the fact that it is very important to evaluate the stealth of the protected code, the method for evaluating the code stealth is sorely lacking. Thus, in this paper, we propose a metric for measuring code stealth. There are many factors causing decrease of code stealth, which include artificial code and unusual behavior. Artificial code is code that does not resemble *raw* compiled code such as encrypted code mentioned earlier. Unusual behavior is the behavior that rarely appears in common software. Examples include self-modification techniques, often used by dynamic obfuscation.

As a first step in evaluating code stealth, in this paper, we propose a method to quantitatively evaluate the *artificiality* of the protected code by means of an N-gram model. We exploit the N-gram model, which is constructed based on a large assembly code corpus, to compute the likelihood estimate for a given assembly code fragment. In a case study, we use the proposed method to evaluate the artificiality of a DRM player routine that is transformed by well-known obfuscation techniques, and discuss the stealth of the routine.

II. Code Artificiality

A. Definition: code

In this paper, we use *code* to mean the whole or part of the program that is described in assembly code. Code is represented as a sequence of assembly instructions. A sequence of n assembly instructions $i_1 i_2 \ldots i_n$ is represented as i_1^n.

The example code that appears in this paper is written in the x86 assembly language [2] using Intel syntax.

B. Definition: artificiality

N-gram models, which are widely used in speech and natural language processing, are used to compute the probability of a sequence of words in the form of an (N-1)-order Markov model. In this study, we use an N-gram model to compute a probability of occurrence of a given code fragment using a corpus made up of a collection of software programs.

$P(i_1^n)$, the probability of a sequence of instructions $i_1^n = i_1 i_2 \ldots i_n$, is computed as follows [3]:

$$P(i_1^n) \approx \prod_{k=1}^{n} P(i_k | i_{k-N+1}^{k-1}) \tag{1}$$

The probability of i_k (the kth instruction) only depends on the previous (N-1) instructions $i_{k-N+1} \ldots i_{k-2} i_{k-1}$. We consider a lower probability $P(i_1^n)$ to imply a more artificial sequence of instructions.

$A(C)$ is the artificiality of the code C that consists of $i_1^n = i_1 i_2 \ldots i_n$. It is defined as follows:

$$Artificiality \quad A(C) = -log_{10} P(i_1^n) \tag{2}$$

The probability is estimated using a corpus of assembly code. $P(i_k|i_{k-N+1}^{k-1})$ can be computed from the maximum likelihood estimation [3]:

$$P(i_k|i_{k-N+1}^{k-1}) = \frac{F(i_{k-N+1}^{k})}{F(i_{k-N+1}^{k-1})} \tag{3}$$

where $F(i_p^q)$ is the number of i_p^q in the corpus. We use a smoothing method, such as absolute discounting [3], to account for sparse data.

III. Case Study

A. Overview

We conducted a case study to evaluate the artificiality of programs obfuscated by methods described in the literature. The N-gram model was constructed using 2,030 open source software applications (e.g., editors, compilers, and games). They were obtained from the collection of applications in Cygwin [4]. These executables are written in PE (Portable Executable) format, and run on the x86 architecture. The size of them extends from 2KB to 17MB. The main resource of this corpus is the list of opcodes of assembly instructions stored in the `.text` section (which holds the program code). The assembly instructions are obtained by `objdump` [5], which uses the linear sweep technique. In this study, only instruction opcodes were collected; in future work we intend to also take operands into account.

We built and applied the N-gram model using SRILM toolkit [6]. TABLE I shows the ten most frequent N-grams (N is varied from 1 to 3) that appeared in the constructed corpus. The values in the table show the ratio of the N-gram's frequency to the total frequency. We can see that the N-grams that include `mov` occur at a high frequency in the corpus, regardless of N.

The target code is a simple DRM player module (hereafter called the `player` module), which is described in [1]. The `player` module has two routines: license checking and decryption. License checking is performed by a simple conditional branch. The decryption routine is applied to encrypted media by means of an XOR operation with a user key and an internal player key. A sample code of the `player` module in the C language is shown in Fig. 1.

TABLE I
FREQUENT N-GRAMS IN THE CORPUS

	1-gram		2-gram		3-gram	
1	mov	44.6	mov-mov	25.0	mov-mov-mov	15.0
2	call	7.56	mov-call	6.85	mov-mov-call	4.97
3	lea	4.77	call-mov	4.24	mov-call-mov	3.87
4	cmp	3.94	lea-mov	2.58	call-mov-mov	2.54
5	je	3.76	nop-nop	2.36	nop-nop-nop	1.86
6	test	3.64	je-mov	2.15	lea-mov-mov	1.53
7	add	3.56	mov-test	1.67	test-je-mov	1.15
8	jmp	3.46	mov-lea	1.66	mov-lea-mov	1.13
9	nop	3.38	test-je	1.66	je-mov-mov	1.11
10	push	2.62	jmp-mov	1.46	mov-test-je	0.94

The values show the ratio of the frequency to the total frequency[%].

```
int play(unsigned int user_key,
         unsigned int encrypted_media[],
         int media_len) {

  int i, code;
  printf("Please enter activation code: ");
  scanf("%d", &code);
  if (code!=ACTIVATION_CODE) {
    fprintf(stderr, "%s!\n", "wrong code");
    return -1;
  }

  *key = user_key ^ player_key;
  for(i=0; i<media_len; i++) {
    unsigned int decrypted =
        *key ^ encrypted_media[i];
    fprintf(audio, "%x\n", decrypted);
    fflush(audio);
  }
  return 0;
}
```

Fig. 1. Sample code of `player` module

B. Evaluation Targets

TABLE II shows an overview of the evaluation targets in this case study. Each code was obtained by obfuscating or optimizing the original code C_o. We first compiled the transformed code into the executable, and then disassembled the `player` module portion. In TABLE II, the term *static obfuscation* means a technique that transforms the code prior to execution, whereas *dynamic obfuscation* means one that transforms the code at run-time. The instructions (opcodes) for each code appear in the corpus. The obfuscated programs were transformed manually except for C_{enca} and C_{encd}, which were automatically transformed by the *Tigress* obfuscation tool [11], [14]. A portion of our experimental data (e.g., the target code and their dump data) is available from our website[1].

Details of the evaluation targets are described below:

C_{opt_1}, C_{opt_2}, C_{opt_s}: Code optimization

C_{opt_1}, C_{opt_2} and C_{opt_s} are optimized versions of C_o, and these are not obfuscated at all. Specifically, the executables of C_{opt_1}, C_{opt_2}, and C_{opt_s} are obtained by applying GCC (the GNU Compiler Collection) optimization options [7] of `-O1`, `-O2`, and `-Os`, respectively. C_{opt_1} is optimized using simple operations that do not require much compilation time. C_{opt_2} is optimized with nearly all supported optimizations that do not involve a space-speed tradeoff. C_{opt_s} is optimized using operations that do not typically increase the code size, as well as further optimizations designed to reduce the code size. C_o, which is compiled with `-O0`, is not optimized at all.

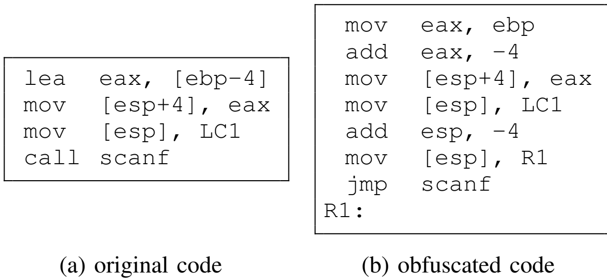
C_{rep}: Replacing with fundamental instructions

C_{rep} is transformed by the obfuscation method proposed in [8]. This method replaces *complicated* instructions in the

[1] http://www.hi.kumamoto-nct.ac.jp/~kanzaki/stealth/SPRO2015/

TABLE II
LIST OF EVALUATION TARGETS

Name	Transformation type	# of opcodes	Outline
C_o	—	60	Original code (without optimization)
C_{opt_1}	Optimization	60	Optimization with `gcc -O1` [7]
C_{opt_2}		64	Optimization with `gcc -O2` [7]
C_{opt_s}		58	Optimization with `gcc -Os` [7]
C_{rep}	Static obfuscation	83	Replace with fundamental instructions [8]
C_{inter}		281	Embedding a specialized interpreter [1]
C_{flat}		86	Control flow flattening [9]
C_{junk}		85	Insertion of junk bytes [10]
C_{enca}		151	Encoding arithmetic [11]
C_{encd}		124	Encoding data [11]
C_{camf}	Dynamic obfuscation	79	Instruction camouflage with instruction selected from the corpus [12]
C_{camfs}		79	Instruction camouflage with instruction selected from C_o [12]
C_{swap}		227	Swapping code fragments [13]
C_{crypt}		267	Swapping code fragments with code encryption [13]
C_{aes}		117	Simple encryption using AES



(a) original code (b) obfuscated code

Fig. 2. Example of obfuscation using fundamental instructions

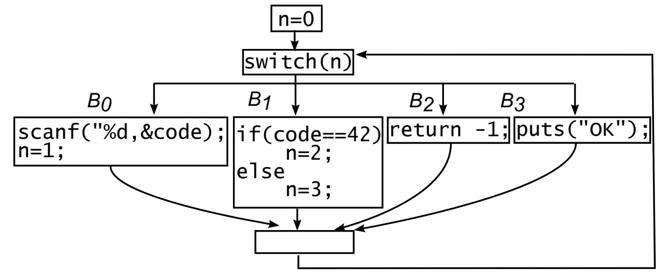


Fig. 3. Example of flattened program

code with *fundamental* instructions (e.g., `mov` and `add`), so that the code consists solely of simple instructions. This makes it more difficult to understand than the original.

Fig. 2 shows an example of obfuscation using fundamental instructions. In this example, `lea` is replaced with `mov` and `add`, `call` is replaced with `add`, `mov`, and `jmp`. C_{rep} is transformed so that the code consists only of these fundamental instructions.

C_{inter}: Embedding a specialized interpreter

C_{inter} is transformed by the obfuscation method proposed in [1]. The method embeds a specialized interpreter in the code, and the program is rewritten based on a customized instruction set architecture. This technique makes it harder for the adversary to design an automated attack that removes the layer of interpretation. C_{inter} has an interpreter engine, and the entire program (`player` module) is written in its customized instructions.

C_{flat}: Control flow flattening

C_{flat} is transformed by the obfuscation method proposed in [9]. This removes the control flow structure (e.g., the nesting of loops and conditional statements) by flattening the control flow graph.

Fig. 3 shows an example of a flattened program (written in the C language). The control flow of the original program is as follows:

1) B_0 is executed.
2) B_1 is executed.
3) B_2 or B_3 is executed according to the result of the conditional branch B_1.

In the obfuscated code, the control flow is flattened using the `switch` statement and variable `n`. The control flow in C_{flat} is flattened using this method.

C_{junk}: Insertion of junk bytes

C_{junk} is the result of inserting junk bytes into C_o. Junk bytes are instructions that have no semantic impact on the program. We transform the conditional jump instructions (e.g., `jne`) using a branch flipping technique [10], and insert junk bytes in the region that is not actually executed. The inserted junk bytes make the program difficult to understand and disassemble. All conditional jump instructions in C_{junk} are transformed, and junk bytes inserted. Each junk byte is randomly selected from C_o.

C_{enca}: Encoding arithmetic

C_{enca} is transformed by the encoding method, which is introduced in [11]. This method replaces integer arithmetic with more complex expressions. The idea is based on a technique which is introduced in Hacker's Delight [15]. Fig. 4 shows an example of obfuscation using this method. In this example, addition operations are replaced with complex expressions

```
z = x + y + w;
```

(a) original code

```
z = (((x ^ y) + ((x & y) << 1)) | w) +
    (((x ^ y) + ((x & y) << 1)) & w);
```

(b) obfuscated code

Fig. 4. Example of encoding arithmetic

```
int c = 42;
if (x == c) ..
```

(a) original code

```
int c = 1848620654;
if (x == -1492092953 * c - 3283795736U) ..
```

(b) obfuscated code

Fig. 5. Example of encoding data

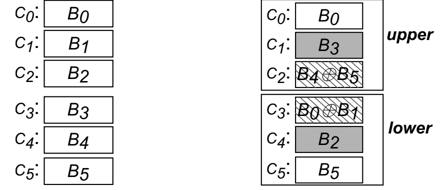
```
pop   edx
cmp   esi, edi
jge   L1
mov   eax, [ebp+12]
```

(a) original code

```
  mov  BYTE [T1], 0x7d
        :
  pop  edx
  cmp  esi, edi
T1:
  je   L1
  mov  eax, [ebp+12]
        :
  mov  BYTE [T1], 0x74
```

(b) camouflaged code

Fig. 6. Example of camouflaged code



(a) original fragments (b) obfuscated fragments

Fig. 7. Example of swapping pieces

using XOR, OR, AND and shift operators. In C_{enca}, all integer arithmetic expressions are obfuscated by this method.

C_{encd}: Encoding data

C_{encd} is transformed by the data encoding method which is introduced in [11]. This method encodes integer variables in the program so that they have a non-standard data representation. The real value of the target variable is replaced with a different integer value, and the real value will be revealed only when needed (e.g., when the value is needed to be output). Fig. 5 shows an example of obfuscation using this method. In this example, the real value of `c` (42) is hidden using this method. In C_{encd}, the integer variables `key` and `ACTIVATION_CODE` (see Fig. 1) are encoded by this method.

C_{camf}, C_{camfs}: Instruction camouflage

C_{camf} and C_{camfs} are *camouflaged* by the method proposed in [12]. Some code fragments in the program are overwritten with dummy instructions, making the original program harder to understand. The program is self-modifying, automatically replacing the dummy instructions with the original ones.

Fig. 6 shows an example of a camouflaged program. In this example, `jge` (its binary code is `0x7d`) is overwritten (camouflaged) with `je` (its binary code is `0x74`). The dummy instruction `je` is restored to the original instruction `jge` at run-time by "`mov BYTE [T1], 0x7d`".

All of the `call`, `cmp`, and conditional jump instructions in C_{camf} and C_{camfs} are camouflaged. The routines which restore the instructions are placed outside `player` module. The dummy instructions in C_{camf} are randomly selected from the instructions that appear in the corpus, whereas the dummy instructions in C_{camfs} are randomly selected from those in C_o.

C_{swap}, C_{crypt}: Swapping code fragments

C_{swap} and C_{crypt} are transformed by the obfuscation method proposed in [13]. This method splits the program into code fragments called *cells*, and runs the cells in order, XORing them with each other. An adversary can only obtain those code fragments that are in cleartext.

Fig. 7 shows an example of the obfuscated program. The original program is split into six cells $C_0, C_1, \ldots, C_5$, and each cell has the original block B_0, B_1, …, B_5 (Fig. 7 (a)). The cells in the obfuscated program are divided into two regions in memory, upper and lower (Fig. 7 (b)). When the program is executed, each cell in upper memory is XORed with a cell in lower memory, so that the original behavior will be performed. At each point during execution, some cells will be in cleartext while others will be hidden. The entire code of C_{swap} is obfuscated by the method described above. C_{crypt} is obfuscated by an extended method, which is combined with an encryption technique to make more of the cells unreadable.

C_{aes}: Simple encryption using AES

C_{aes} is encrypted by means of the AES (Advanced Encryption Standard) algorithm. The entire code is overwritten with data obtained by encrypting the original machine code. The original code of C_{aes} is obtained at run-time by a decryption routine using the self-modifying technique. The decryption routine itself is located outside of C_{aes} and is executed before C_{aes} is called. The disassembled code of C_{aes} is meaningless as machine code, because C_{aes} is just encrypted data.

C. Results

We constructed the N-gram model and measured the artificiality of each target code described in Section III-B. N was varied from 1 to 3. The results are shown in Fig. 8. The artificiality of the optimized code C_{opt_1}, C_{opt_2}, and C_{opt_s} is similar to that of the original code C_o. In contrast, the artificiality of C_{inter}, C_{swap}, C_{crypt}, and C_{aes} are very high.

Fig. 9 shows the artificiality of each evaluation target in the case $N = 3$. The vertical axis represents the artificiality,

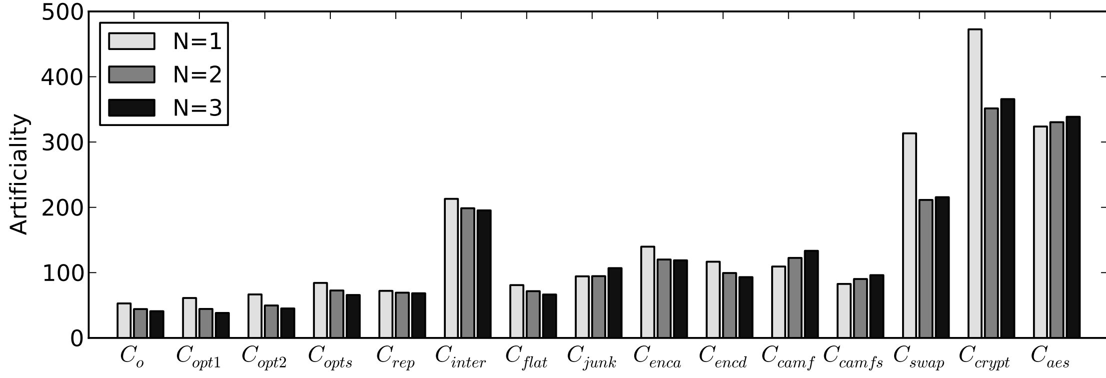


Fig. 8. Results for the artificiality of the target code

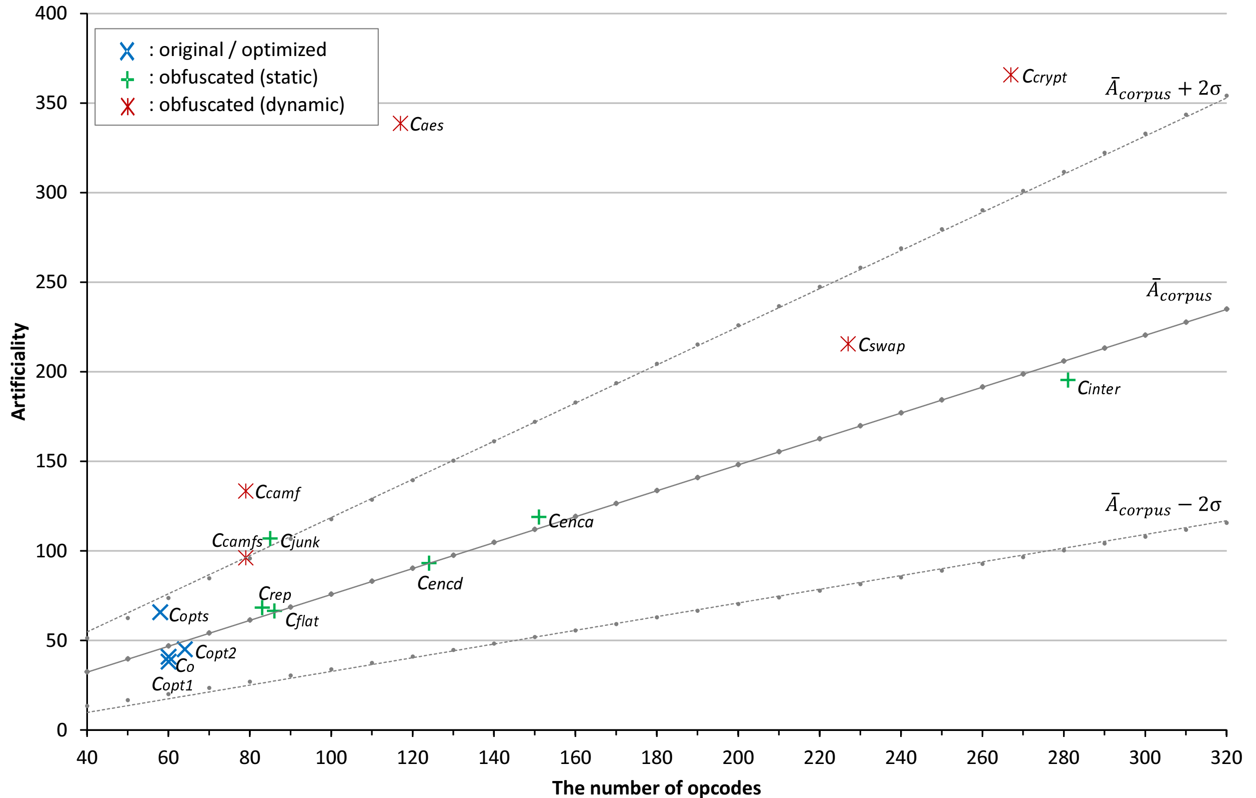


Fig. 9. Relation between the artificiality and the number of opcodes ($N = 3$)

and the horizontal axis represents the number of opcodes (the length of the code fragment). For comparison, we also examined the average ($\overline{A}_{corpus}$) and the standard deviation (σ) of the artificiality values of unobfuscated code samples in the corpus. They were obtained as follows:

1) A code fragment (i.e., a sequence of instructions) that consists of m instructions is selected randomly from the corpus. This is repeated 100,000 times. The obtained fragments are denoted as $c_1, c_2, \ldots, c_{100,000}$.
2) The artificiality values of these code fragments, $A(c_1)$, $A(c_2)$, $\ldots$, $A(c_{100,000})$ are calculated.
3) We measure the average and the standard deviation of $A(c_1)$, $A(c_2)$, $\ldots$, $A(c_{100,000})$, which are denoted as $\overline{A}_{corpus}(m)$ and $\sigma(m)$, respectively.

We measured $\overline{A}_{corpus}(m)$ and $\sigma(m)$, varying m from 40 to 320 in intervals of 10. In Fig. 9, the solid line shows $\overline{A}_{corpus}$ and the dotted lines show $\overline{A}_{corpus} + 2\sigma$ and $\overline{A}_{corpus} - 2\sigma$, which were obtained by the method of least squares. The original code C_o, the optimized code and many obfuscated code are within $\overline{A}_{corpus} \pm 2\sigma$. However, some obfuscated code, such as C_{junk}, C_{camf}, C_{camfs}, C_{crypt}, and C_{aes}, are not within $\overline{A}_{corpus} \pm 2\sigma$.

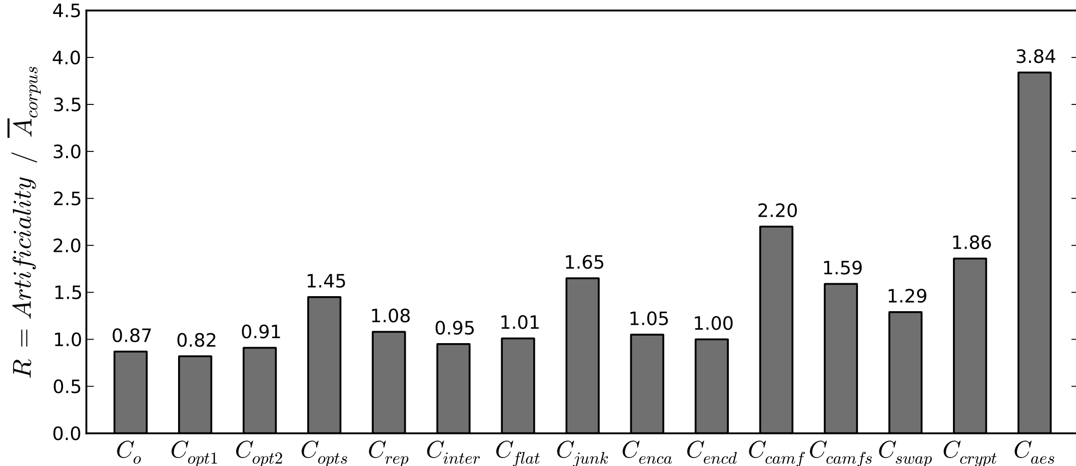


Fig. 10. Ratio between artificiality and $\overline{A}_{corpus}$ ($N = 3$)

In addition, as seen in Fig. 9, artificiality $A(C)$ is highly dependent on the number of opcodes in the code. To facilitate a comparison of artificiality, we introduce the normalized artificiality $R(C)$, which is the ratio between artificiality and $\overline{A}_{corpus}$:

$$R(C) = \frac{A(C)}{\overline{A}_{corpus}(length(C))} \tag{4}$$

where $length(C)$ is the number of opcodes of C. Fig. 10 shows $R(C)$ for each evaluation target in the case $N = 3$. We can see from this graph that $R(C_{junk})$, $R(C_{camf})$, $R(C_{camfs})$, $R(C_{crypt})$, and $R(C_{aes})$ are relatively large.

D. Analysis

In many cases, the artificiality of optimized code and code obfuscated by static obfuscation differs little from $A(C_o)$, regardless of N. This is because the transformations applied to the code leave many code fragments of the program unchanged. The artificiality of C_{inter}, which was obfuscated by static obfuscation, seems larger than that of other code obfuscated statically (Fig. 8). This is because the number of opcodes in C_{inter} is large, because the code includes an interpreter engine. From the results shown in Fig. 9 and Fig. 10, we can see that C_{inter} is not so artificial if we take into account the number of opcodes. In contrast, from Fig. 9 and Fig. 10, C_{junk} tends to be artificial. We believe that the inserted meaningless instructions increase the unusualness of the code.

From the results shown in Fig. 8, Fig. 9, and Fig. 10, it can be seen that most code subjected to dynamic obfuscation tends to be artificial. First, the artificiality of the camouflaged code C_{camf} is high. This is because some instructions are overwritten with dummy instructions, which increases unusual code fragments. The artificiality of C_{camfs} is not particularly high in comparison with C_{camf}. This is because C_{camfs} has fewer unusual code fragments, as the dummy instructions are selected from C_o, which consists of common instructions. This fact is useful in selecting dummy instructions when camouflaging the code. The artificiality of C_{crypt} is also high, because parts of the code are XORed and encrypted, which produce many unusual code fragments. The artificiality of C_{aes}, which is entirely encrypted, is very high, as the disassembled code is meaningless as assembly code.

The results of this case study show that optimization and static obfuscating transformations (e.g., replacing with fundamental instructions, embedding a specialized interpreter, flattening control flow and encoding arithmetic/data) have little effect on artificiality. However, dynamic obfuscating transformations (e.g., instruction camouflage and code encryption), or a technique that inserts junk code fragments into the program, tend to increase the artificiality, which may have a significant impact on the stealth of the code.

E. Discussion

We next discuss three potential issues with our experimental design.

First of all, our case study uses only one target program, a small, artificial, prototypical cracking target [1]. The reason is that many of the transformations had to be applied by hand, since appropriate obfuscation tools do not exist in the open community, and hence applying the transformations to multiple targets would have been too time-consuming. In future work our goal is to obfuscate various types of programs containing different types of security sensitive code, and examine the resulting code artificiality.

Second, we ignored operands in our N-gram model. Thus, code fragments which are artificial due only to unique operands will not be detected. For example, we cannot detect unusual branch addresses which may be caused by the control-flow obfuscation or instruction camouflage. In future work we intend to extend our N-gram model to take operands (especially branch addresses) into account. It would be interesting to, for each branch, classify it as direct/indirect,

forward/backward or conditional/unconditional in the N-grams to see if such features can help detect flattened code and other control-flow transforms.

Third, for most of the methods we applied the obfuscating transformations by hand. It is possible that the level of artificiality may be affected by the person who obfuscates the program. To avoid such bias in the experimental results, we are actively developing an appropriate automatic obfuscation tool, Tigress [11], [14]. We believe that varying the amount of obfuscation added by the tool can make the results more interesting.

IV. Related Work

Some methods for evaluating the difficulties of program analysis have been proposed. For example, the program complexity metric [16] and queue-based mental simulation model [17] are used to quantify the difficulty of understanding a program. There is also a study which aims to assess the difficulty of understanding and modifying obfuscated code through controlled experiments involving human subjects [18]. In addition, a survey reported in [19] investigates the effectiveness of dynamic obfuscation methods using *visibility* and *exposure* metrics. The proposed method aims to measure the stealth of code, which has not been discussed before.

The technique of analyzing the machine code based on a probabilistic approach such as N-grams has been used in studies that aim to detect or classify malware. Karim et al. proposed a method for measuring the similarity of malware based on the cosine similarity of N-grams of opcodes [20]. In addition, Lyda et al. developed a method for identifying malware that is packed or encrypted using the entropy of machine code [21]. These methods quantitatively evaluate the low-level characteristics of code, in common with the proposed method. Whereas previous methods use these characteristics to judge the state or similarity of malware, the proposed method exploits them to measure the artificiality of code, which is needed to evaluate the stealth of protected code.

V. Conclusion

This paper proposes a method for evaluating the artificiality of protected code by means of an N-gram model. The proposed evaluation method estimates one aspect of the stealth of protected code, that is, the degree to which protected code can be distinguished from unprotected code.

In the case study, we applied the proposed method to evaluate the artificiality of various programs that had been obfuscated by well-known obfuscation techniques. The results showed that static obfuscating transformations (e.g., transformation of the control flow of the program) have little effect on artificiality. On the other hand, dynamic obfuscating transformations (e.g., code encryption), or a technique that inserts junk code fragments, tend to increase the artificiality, making a significant impact on the stealth of the code.

We are currently developing an obfuscation tool that automatically outputs variant obfuscated programs. This tool will allow us to conduct further case studies that consider multiple input programs and take into account both opcodes and operands. We will furthermore extend our study to measure stealth based on dynamic traces of obfuscated programs. Trying a different disassembler which uses the recursive traversal technique would also be interesting.

Acknowledgment

This work was supported by Grant-in-Aid for Scientific Research (Grant Number 26330094), Japan Society for the Promotion of Science (JSPS).

References

[1] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Program Protection*. Addison-Wesley Professional, 2009.

[2] *Intel 64 and IA-32 Architectures software developer's manual vol.1 : Basic Architecture*, Intel Corporation, http://www.intel.com/products/processor/manuals/ (accessed: Jan. 2015).

[3] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 2nd ed. Pearson Prentice Hall, 2008.

[4] "The Cygwin project," http://www.cygwin.com/, (accessed: Jan. 2015).

[5] "GNU binary utilities," https://sourceware.org/binutils/docs/binutils/, (accessed: Jan. 2015).

[6] "SRILM – the SRI language modeling toolkit," http://www.speech.sri.com/projects/srilm/, (accessed: Jan. 2015).

[7] Free Software Foundation, "GCC online documentation," http://gcc.gnu.org/onlinedocs/, (accessed: Jan. 2015).

[8] M. Mambo, T. Murayama, and E. Okamoto, "A tentative approach to constructing tamper-resistant software," in *Proc. 1997 New Security Paradigm Workshop*, Sep. 1997, pp. 23–33.

[9] C. Wang, J. Hill, J. C. Knight, and J. W. Davidson, "Protection of software-based survivability mechanisms," in *Proc. 2001 International Conference on Dependable Systems and Networks*, 2001, pp. 193–202.

[10] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. 10th ACM Conference on Computer and Communications Security*, Oct. 2003, pp. 290–299.

[11] C. Collberg, "The Tigress diversifying C virtualizer," http://tigress.cs.arizona.edu/, (accessed: Jan. 2015).

[12] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto, "Exploiting self-modification mechanism for program protection," in *Proc. 27th IEEE Computer Software and Applications Conference*, Dallas, USA, Nov. 2003, pp. 170–179.

[13] D. W. Aucsmith, *Tamper Resistant Software: An Implementation*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1996, vol. 1174, pp. 317–333.

[14] C. Collberg, S. Martin, J. Myers, and J. Nagra, "Distributed application tamper detection via continuous software updates," in *Proc. 28th Annual Computer Security Applications Conference*, Orlando, Florida, Dec. 2012, pp. 319–328.

[15] H. S. Warren, *Hacker's Delight (2nd Edition)*. Addison-Wesley Professional, 2012.

[16] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Technical Report of Dept. of Computer Science, University of Auckland, New Zealand, Tech. Rep. 148, 1997.

[17] M. Nakamura, A. Monden, T. Itoh, K. Matsumoto, Y. Kanzaki, and H. Satoh, "Queue-based cost evaluation of mental simulation process in program comprehension," in *Proc. 9th IEEE International Software Metrics Symposium (METRICS2003)*, Sep. 2003, pp. 351–360.

[18] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "Towards experimental evaluation of code obfuscation techniques," in *Proc. the 4th ACM Workshop on Quality of Protection*, Alexandria, Virginia, USA, 2008, pp. 39–46.

[19] N. Mavrogiannopoulos, N. Kisserli, and B. Preneel, "A taxonomy of self-modifying code for obfuscation," *Computers & Security*, vol. 30, no. 8, pp. 679–691, 2011.

[20] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *European Research Journal of Computer Virology*, vol. 1, no. 1-2, pp. 13–22, Nov. 2005.

[21] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security and Privacy*, vol. 5, no. 2, pp. 40–45, 2007.