Pinpointing and Hiding Surprising Fragments in an Obfuscated Program

Yuichiro Kanzaki[*][†]
kanzaki@
kumamoto-nct.ac.jp

Clark Thomborson[†]
c.thomborson@
auckland.ac.nz

Akito Monden[‡]
monden@
okayama-u.ac.jp

Christian Collberg[§]
collberg@gmail.com

[*]National Institute of Technology, Kumamoto College, Japan
[†]Department of Computer Science, University of Auckland, New Zealand
[‡]Graduate School of Natural Science and Technology, Okayama University, Japan
[§]Department of Computer Science, University of Arizona, USA

ABSTRACT

In this paper, we propose a *pinpoint-hide* defense method, which aims to improve the stealth of obfuscated code. In the pinpointing process, we scan the obfuscated code in a few small code fragment level and identify all *surprising* fragments, that is, very unusual fragments which may draw the attention of an attacker to the obfuscated code. In the hiding process, we transform the pinpointed surprising fragments into unsurprising ones while preserving semantics. The obfuscated code transformed by our method consists only by unsurprising code fragments, therefore is more difficult for attackers to be distinguished from unobfuscated code than the original. In the case study, we apply our pinpoint-hide method to some programs transformed by well-known obfuscation techniques. The result shows our method can pinpoint surprising fragments such as dummy code that does not fit in the context of the program, and instructions used in a complicated arithmetic expression. We also confirm that instruction camouflage can make the pinpointed surprising fragments unsurprising ones, and that it runs correctly.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]; D.2.8 [**Metrics**]

Keywords

Software Protection, Code Obfuscation, Program Analysis, Code Stealth, N-gram

PPREW-5, December 08 2015, Los Angeles, CA, USA
© 2015 ACM. ISBN 978-1-4503-3642-0/15/12…$15.00
DOI: http://dx.doi.org/10.1145/2843859.2843862

1. INTRODUCTION

Code obfuscation techniques are widely used for protecting software products against MATE (man-at-the-end) attacks, that is, attacks by an end user who has access to software itself and attempts to steal assets in the program. The objective of code obfuscation is to make a program more difficult to understand for adversarial humans, while preserving program semantics for its desired users. Many methods of obfuscation have been proposed [8,9,15], and there are some implementations, e.g. *Tigress* [6, 7] and *DonQuixote* [17].

In this paper, we view obfuscation as a defensive strategy. We assume that a defender has obfuscated a program, in order to mitigate the risk that an attacker could discover a license key or other confidential property of the program. In this security model, the defender wants to assess the *strength* of their obfuscation. Informally, the strength of an obfuscation is any measure of the effort required for an adversary to understand some security-critical aspect of the code well enough to subvert it. License-checking code, any use of cryptographic keys, or any use of cryptographic hashes, are examples of security-critical aspects. In prior research, various approaches have been taken to the measurement of strength, such as evaluating standard metrics of code complexity [9], queue-based mental simulation models [16], and controlled experiments involving human subjects [5].

Defenders also want to measure the *stealth* of an obfuscation, that is, the degree to which obfuscated code can be distinguished from unobfuscated code [8]. Obfuscations with high stealth cannot be readily located by automated means, thereby making these obfuscations more resilient to a *locate-alter-test* attack (i.e., an attack which goes through locating the protected part in the binary executable, altering the located part, and testing the result of the altered code) – because such attacks must, in general, be narrowly focused on small segments of code in order to be feasible.

We coin the phrase *pinpoint-alter-test* to describe an attack in which just a few instructions are targeted for adversarial modification on an experimental basis. We coin the phrase *pinpoint-hide* to describe a defense strategy of pinpointing all surprising fragments, then hiding these fragments by transforming these (while preserving semantics) into unsurprising ones.

In this paper, we propose a *pinpoint-hide* defense method, which is based upon a formal definition of *surprise* and ultimately upon an n-gram model of a large corpus of assembly-language programs. Roughly speaking, a surprising code fragment is any fragment which appears at most rarely in the corpus. Because surprising fragments may draw the attention of an attacker to obfuscated code, it is in the defender's interest to use a method which produces only unsurprising code.

In our case study, we first pinpoint the surprising fragments in programs that were obfuscated by well-known obfuscation techniques. We find that some of these techniques are not stealthy with respect to our pinpointing method. We then show how the surprising fragments in these obfuscations can be hidden, using a method that is based on instruction camouflage [13], thereby increasing their stealth.

2. PRELIMINARY

2.1 Code fragment and corpus

In this paper, we use *code fragment* (or simply *fragment*) to mean a sequence of x86 assembly instructions. In our prototype, we use only a few features of each instruction: its opcode mnemonic and types of operands. We use IDA's operand classifier [11] shown in Table 1. For example, we encode "`mov eax, [ebp-10h]`" as "`mov14`" because it has one type-1 (register) operand and one type-4 (memory reference) operand. Literal data and alignment pseudo-ops that appear in the code sections are encoded as the pseudo-instructions `DATA` and `ALIGN`, respectively.

We write i_j^n to denote the sequence of features of a length-n fragment which starts at j-th instruction of a program. We call any length-n fragment an n-gram. When n=1, we omit the superscript, writing i_j for a single instruction.

The corpus we use as a basis for our model of surprise is composed of the disassembled code sections (`.text` sections) of 3,071 Windows executables written in PE (Portable Executable) format. We obtained these executables from the applications in a recent Cygwin [1] distribution. The applications are of various types such as shells, editors, compilers and games. Their size is highly variable, from 4KB to 32.9MB. We used IDA v6.8 [11] to disassemble executables. There are approximately 1.30×10^8 instructions in our corpus, when it is disassembled by IDA.

Table 1: Operand types classified by IDA.

Value (hex)	Description
0	no operand
1	registers
2	direct memory data reference
3	memory reference using register contents
4	memory reference using register contents (with displacement)
5	immediate value
6	immediate far address
7	immediate near address
8-D	processor specific type

2.2 Fragment surprisal

To define surprise of an instruction, we focus on its precedence instructions to capture its context so that the same instruction in different contexts could have different surprising measures. We define the conditional surprisal $S()$ of the last instruction in fragment i_j^n as this instruction's self-information with respect to our corpus:

$$S(i_j^n) = -log_2 P(i_{j+n-1} | i_j^{n-1}) \quad \text{bits} \qquad (1)$$

A conditional probability $P(i_{j+n-1}|i_j^{n-1})$ of the last instruction in fragment i_j^n is computed from the maximum likelihood estimation $c(i_j^n)/c(i_j^{n-1})$, where $c()$ maps n-grams onto frequency counts in the corpus. In this notation, a 0-gram is the starting-point of an instruction, that is, $c(i_j^0)$ represents the total number of instructions in the corpus. We *smooth* the probabilities, so that any fragment which does not occur in the corpus are assigned a very high (but non-infinite) surprisal value. In our case study, we use Katz smoothing which is implemented in the SRILM toolkit [2].

We compute a *fragment surprisal* $s(i_j^n)$ by summing the conditional surprisals of all prefixes, and by normalizing by the length of the fragment:

$$s(i_j^n) = \sum_{1 \leq m \leq n} S(i_j^m)/n \quad \text{bits} \qquad (2)$$

The normalization allows us to compare surprisals for our n-gram models. Intuitively, a fragment surprisal represents the degree of the rarity or unusuality when the fragment occurs in an unobfuscated program. For instance, based on our corpus, s(`mov14 test11 jz70`) is calculated as 2.437, while s(`inc10 nop00 dec10`) is calculated as 12.95. We can see from these values that occurring the fragment `inc10-nop00-dec10` in an unobfuscated program is much more surprising than occurring the fragment `mov14-test11-jz70`.

The surprisal of an obfuscated code is evaluated with respect to a disassembled corpus, so its value will depend at least slightly on the disassembler. The IDA disassembler uses a recursive traversal to discover the reachable blocks in the text segment of an executable file. Any code which is *not* reached by the recursive traversal is disassembled into `DATA` (with e.g. the `db` pseudo-op denoting a one-byte literal). The `DATA` pseudo-instruction rarely, if ever, follows non-branching instructions in unobfuscated code. In our case study, we confirmed that the anti-disassembly feature of some obfuscations resulted in such 2-grams – which are easily recognized as surprising because of their low probability of occurrence.

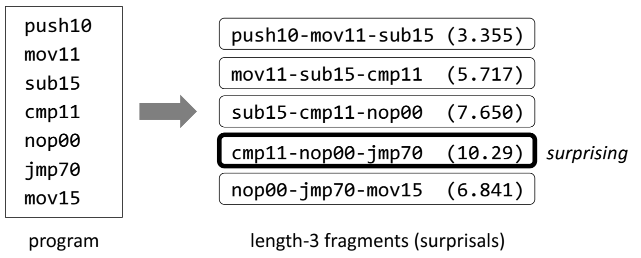

push10
mov11
sub15
cmp11
nop00
jmp70
mov15
push10-mov11-sub15 (3.355)
mov11-sub15-cmp11 (5.717)
sub15-cmp11-nop00 (7.650)
cmp11-nop00-jmp70 (10.29)
surprising
nop00-jmp70-mov15 (6.841)
program
length-3 fragments (surprisals)

Figure 1: Concept of pinpointing surprising fragments.

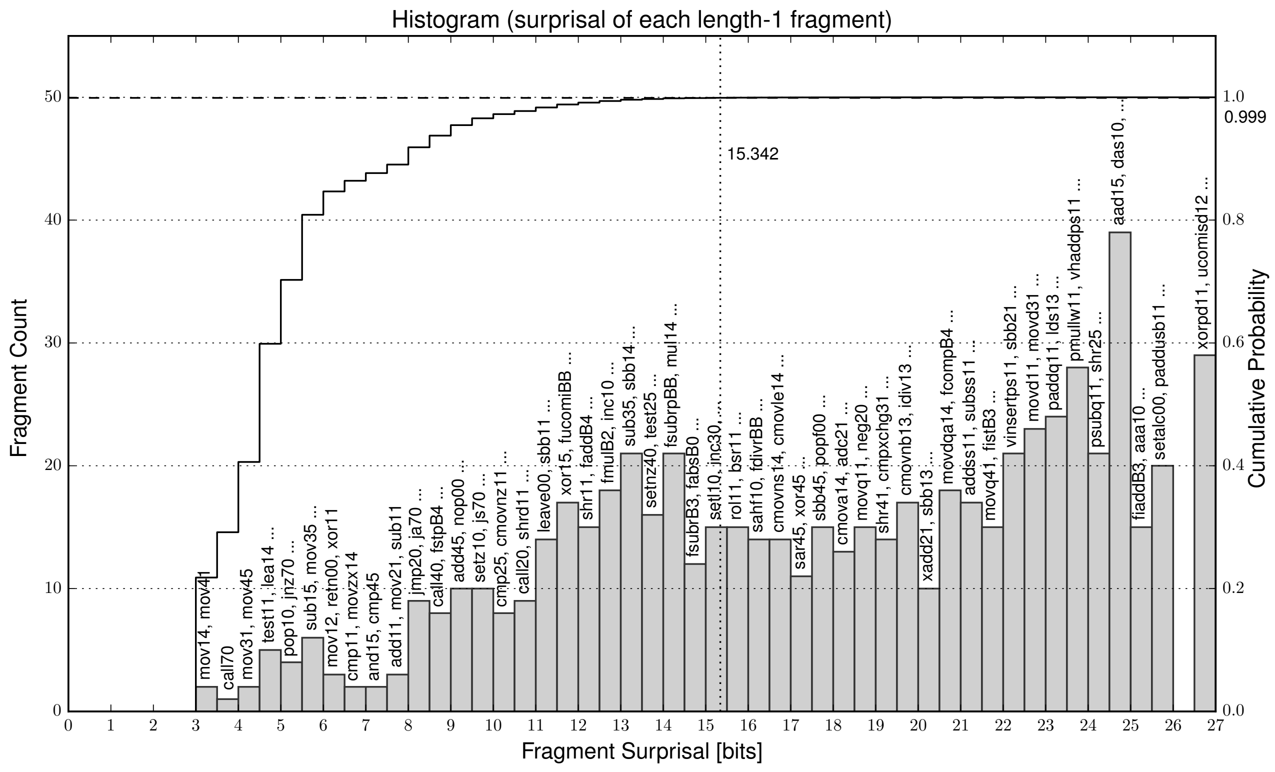


Figure 2: Histogram of 1-gram (single instruction) surprisals.

The artificiality, or overall probability of occurrence, of an entire program is $\sum_j S(i_j^n)$. The sum is taken over all of the fragments in the program, including the first few fragments for which a lower-order model is required. In our prior work [12], we found that some methods of obfuscation, in particular code-encryption and inserting junk code, significantly increase artificiality and thereby decrease stealth. In this article we extend our prior work by the use of pinpointing and hiding, as described below.

3. PINPOINTING AND HIDING SURPRISING FRAGMENTS

3.1 Pinpointing surprising fragments

Pinpointing is the process of identifying all surprising fragments in a program. In this process, we first obtain each length-n fragment in the program, for a given value of n. Specifically, if the program is composed of instructions i_1^k, we obtain $k - n + 1$ fragments, i_1^n, i_2^n, ..., i_{k-n+1}^n. We then compute the fragment surprisals of each fragment, and any fragment whose surprisal is the threshold value or more is considered to be surprising fragment. Figure 1 illustrates a concept of pinpointing surprising fragments. In this example, the five length-3 fragments are obtained from the program which has seven instructions, and the fragment surprisals of each fragment are computed. Assuming that the threshold values of surprisal is 10.00, the fragment `cmp11-nop00-jmp70` is considered to be surprising fragment because its surprisal is above the threshold. Our purpose in the pinpointing process is to scan an obfuscated program in a few small code fragment level in order to pinpoint the exact part that may cause to decrease the stealth. For this reason, n is varied from 1 to 3 in our case study.

Next, we explain how to determine the threshold values of surprisal. We first sort the unique fragments in our corpus, in order of decreasing surprisal with respect to an n-gram model of the corpus. We then plot the cumulative probability distributions. For example, Figure 2 shows the histogram of instruction frequencies (1-grams) in our corpus. The horizontal axis represents the fragment surprisal, while the vertical axes plot the fragment count and the cumulative probability. Some example fragments are also shown at each bin.

The frequency distribution of n-grams in our corpus is very skewed. As indicated by the shaded bars in Figure 2, two instructions (`mov14`, `mov41`) have surprisal in the range 3.0 to 3.5 bits; one instruction (`call70`) with surprisal in the range [3.5,4.0); ... fifteen instructions with surprisal in the range [15.0, 15.5); ... and twenty-nine instructions with surprisal in the range [26.5, 27.0). The vertical dotted line is at the 99.9% point in the cumulative-probability line.

In our case study, we use $\theta = 0.999$ as the threshold of our surprisal detector. The false-positive rate of our pinpoint-detector is thus approximately $1 - \theta = 0.1\%$. Note: the false-positive rate of our pinpoint detector is likely to be somewhat higher, if it were evaluated against a different corpus of unobfuscated code than the one it was trained on. The robust estimation of a false-positive rate for our proposed pinpoint detector is outside the scope of this paper – our goal is to define a plausibly-accurate method of pinpoint-detection, and to perform an initial estimate of its accuracy and feasibility.

Table 2: Information about the fragments and surprisals in the corpus (where θ=0.999).

	$n = 1$	$n = 2$	$n = 3$
# of fragment patterns	641	641^2	641^3
The most frequently-observed fragments	`mov14`	`mov31-call70`	`pop10-pop10-pop10`
Threshold value of surprisal $t^n_{0.999}$	15.342	11.125	8.9523
% of fragment patterns below threshold	35.4%	3.48%	0.131%
Example fragment corresponding to $t^n_{0.999}$	`inc30`	`and15-fldB3`	`movsx14-cwde10-mov12`

Table 2 presents some additional information about the fragments and surprisals in our corpus, where n=1,2,3. In particular, we observed 641 unique 1-grams in our corpus. Slightly more than one-third of these 1-grams occur in θ = 99.9% of the instructions in our corpus. The other 64.6% of the 1-grams have an above-threshold surprisal of t^1_θ = 15.342 bits or more. Here we have introduced the notation t^n_θ to denote the surprisal, in bits, of the minimally-surprising fragment in an n-gram model with detection threshold θ. The "long right tail" skew is even more pronounced in our 2-gram and 3-gram models. Only approximately 344,000 (0.131%) of the $641^3 \approx 263$ million 3-grams in our 3-gram model are unsurprising.

3.2 Hiding surprising fragments

Hiding is the process of replacing a surprising fragment by one or more non-surprising fragments, while preserving program semantics. In our case study, we use an iterative and heuristic hiding process of selecting a semantically-equivalent replacement i^m_j for a first surprising fragment i^n_j, re-evaluating surprisals starting at i^n_{j-n+1}, then hiding all subsequent surprises. Note that we may replace a fragment of length n by a fragment of length $m \neq n$.

A re-evaluation of immediately-preceding fragments is necessary in our hiding method, because any change to the first instruction in a fragment will modify the suffix of the n-1 preceding fragments in an n-gram model. Because of these re-evaluations, our heuristic method may fail to make forward progress on some codes. However, we have not observed such pathological behavior in our case study. The development of an efficient hiding algorithm is outside the scope of the article. Instead, we focus on establishing its desirability and plausibility as a method for increasing the stealth of an obfuscated code.

Hiding is not a novel concept. Specific hiding methods have been called "instruction camouflage" [13] and "replacing with fundamental instructions" [14] in prior work on obfuscation. Hiding has also been studied in the context of steganography, where binary code has been used as a cover-text for a secret message [3]. To our knowledge, the earliest academic reference to what we are calling a hiding transform are Wayner's "mimic functions" [19]. We believe we are the first to use a precise method ("pinpointing") to identify specific fragments which should be hidden.

4. CASE STUDY

4.1 Target programs

We conducted a case study to pinpoint and hide surprising fragments in obfuscated programs. Figure 3 presents the C-language source of our case-study program P_o. This program consists of a single function which returns a 0/1 value to indicate whether or not a license is valid. If the license has expired, the console user is prompted to enter an activation code. In its present form, this function implements a "Maginot license" because it is readily bypassed by an attacker who disassembles it in order to discover which byte encodes its literal `-1`, then modifies this byte so that it encodes `0`. However, if this function and its callsites are stealthily obfuscated, the attacker must allocate some additional resource to pinpoint its critical opcodes and operands.

```
int checklicense(void) {
  time_t current_time;
  int code;
  time(&current_time); /* set current time */
  if (current_time > mktime(&EXPIRE_TIME)) {
    printf("Your license is expired.\n");
    printf("Enter activation code: ");
    scanf("%d", &code);
    if(code == ACTIVATION_CODE) {
      renewlicense();
    } else {
      printf("Wrong code.\n");
      return -1; /* failure */
    }
  }
  return 0; /* success */
}
```

Figure 3: P_o: Original program.

We obfuscated P_o in five different ways, obtaining P_{opaq}, P_{vir}, P_{flat}, P_{enca}, and P_{encd}. The obfuscating transforms are described briefly below. To avoid introducing bias from a manual obfuscation, and to have a reproducible experimental result, we used Version 2.0 of the *Tigress* obfuscation tool [6,7]. We first compiled the obfuscated code into an executable using GCC, then we disassembled the target function portion using IDA. All of the programs were compiled without optimization, because the optimizer may reverse the applied obfuscation. The target programs and some experimental data used in this case study are available from our website[1].

P_{opaq}: **Insertion of opaque predicates**
An opaque predicate [10] is a conditional branch on a condition whose truth-value is obscure to the attacker. P_{opaq} has ten opaque predicates which preserve the control-flow of P_o while rendering it more difficult for an attacker to understand.

P_{vir}: **Function virtualization**
Function virtualization is the transformation of code into an obscurely-defined bytecode. The text segment of P_{vir} is a bytecode interpreter which takes the form of

[1]http://www.hi.kumamoto-nct.ac.jp/~kanzaki/stealth/PPREW5/

```
int ACTIVATION_CODE = 1848620654U;
          :
if(code == (int )(-1492092953 * ACTIVATION_CODE -
     3283795736U))
          :
```

Figure 4: Obfuscated equality test of `code` with an encoded `ACTIVATION_CODE`.

a `switch` statement that is executed for each bytecode in the data segment of P_{vir}.

P_{flat}: **Control-flow flattening**
Control-flow flattening is the replacement of normal control-flow structures (e.g. nested loops, if-then-else nests) by a `switch` statement in an indefinite loop. The `switch` in P_{flat} has eleven cases.

P_{enca}: **Encoding arithmetic**
Integer arithmetic can be obscurely recoded into more complex expressions. In P_{enca}, the conditions of the `if` statements are transformed by methods explained in a popular book on programming tricks and techniques [18].

P_{encd}: **Encoding data**
The values stored in integer variables can be encoded so that their meaning is obscured. In P_{encd}, the integer literal `ACTIVATION_CODE` (see Figure 3) is encoded. It is decoded before equality-testing with `code`, using the code shown in Figure 4.

We note that encoding data is not a strong obfuscation for P_o, unless additional obfuscating transforms are effectively obscuring the uses of its literals `ACTIVATION_CODE`, `0`, and `-1`.

4.2 Result of pinpointing surprising fragments

We pinpointed surprising fragments of each target program described in Section 4.1. We set the detection threshold θ to 0.999, and n was varied from 1 to 3.

Table 3 shows the number of instructions, basic blocks, and pinpointed surprising fragments of each program. We note that P_o, P_{flat}, and P_{encd} have no surprising fragments.

P_{vir} has two surprising fragments for n=3; we consider this to be an inadequately-strong detection signal by the following argument. This code has 318 instructions. Our detector has a false-positive rate of approximately $1 - \theta = 0.001$, so an unobfuscated code of this length with 318 instructions would have (by a Poisson approximation) two or more pinpoint-detections with probability approximately $1 - e^{-.318}(1 + 0.318) = 0.04$. Any particular adversary may, or may not, be motivated to investigate a detection signal that has a 4% false-positive rate. In our case study, we have somewhat arbitrarily aimed for 99.9% accuracy, so we do not consider the two surprising fragments of P_{vir} to be a reliable detection of an obfuscated code.

Table 3: The number of surprising fragments.

name	# of insts.	# of basic blocks	# of surprising fragments		
			$n=1$	$n=2$	$n=3$
P_o	33	6	0	0	0
P_{opaq}	108	29	0	4	10
P_{vir}	318	60	0	0	2
P_{flat}	157	44	0	0	0
P_{enca}	101	6	4	0	5
P_{encd}	36	6	0	0	0

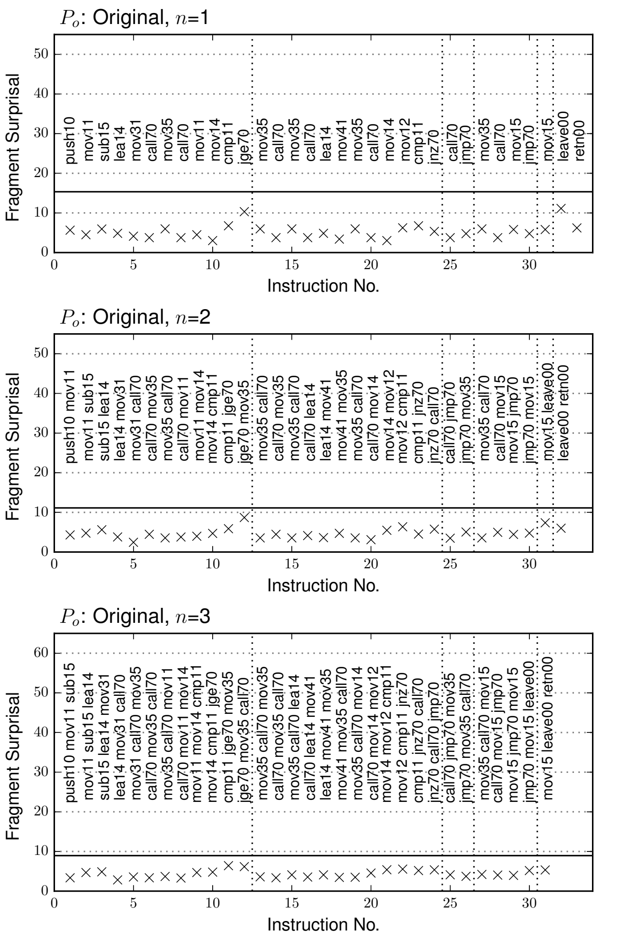


Figure 5: Distribution of surprisal of each fragment in the original version P_o for n=1, 2, 3.

P_{opaq} and P_{enca} have 10 and 5 (resp.) surprising fragments for n=3. These obfuscated programs are reliably detected (with confidence > 99.9%) by our n=3 model, when we use a Poisson approximation to estimate the frequency of false-positive surprises in these programs. P_{opaq} is also surprising for n=2, and $P_{enca}$is surprising for n=1. To determine the success of our pinpointing method, we compare the surprisals of all fragments of P_o, P_{opaq}, and P_{enca}.

Our visualizations of the surprisals of P_o, P_{opaq}, and P_{enca} are shown in Figure 5, Figure 6, and Figure 7, respectively. The horizontal axis represents the instruction number, while the vertical axis plots the fragment surprisal. The horizontal solid line shows the threshold value of surprisal $t^n_{0.999}$, and the vertical dotted lines represent the boundaries of basic blocks. Crosses indicate fragments which appear in P_o(i.e. are unobfuscated), circles indicate fragments that have been

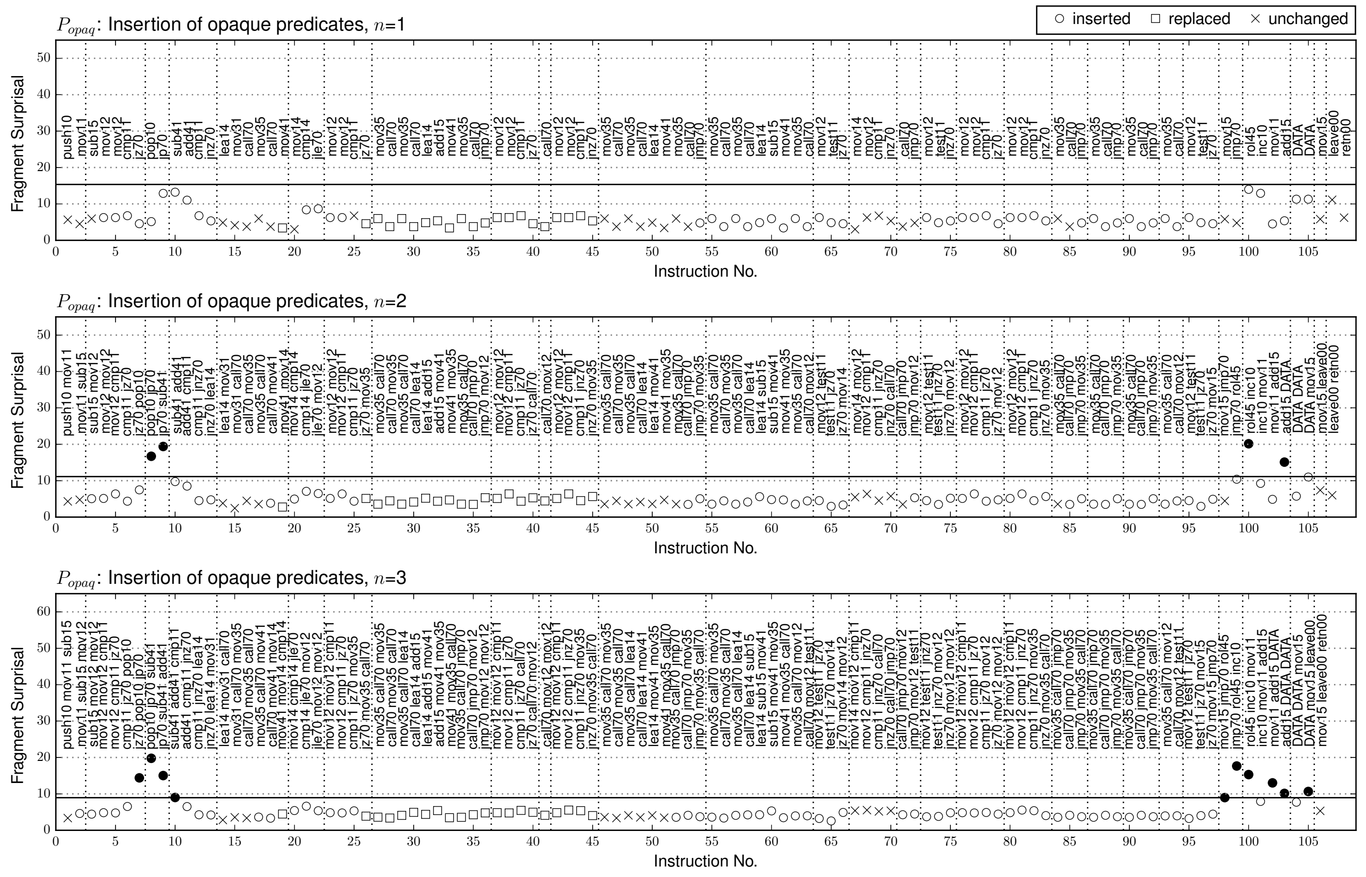


Figure 6: Distribution of surprisal of each fragment in P_{opaq} (insertion of opaque predicates) for n=1, 2, 3.

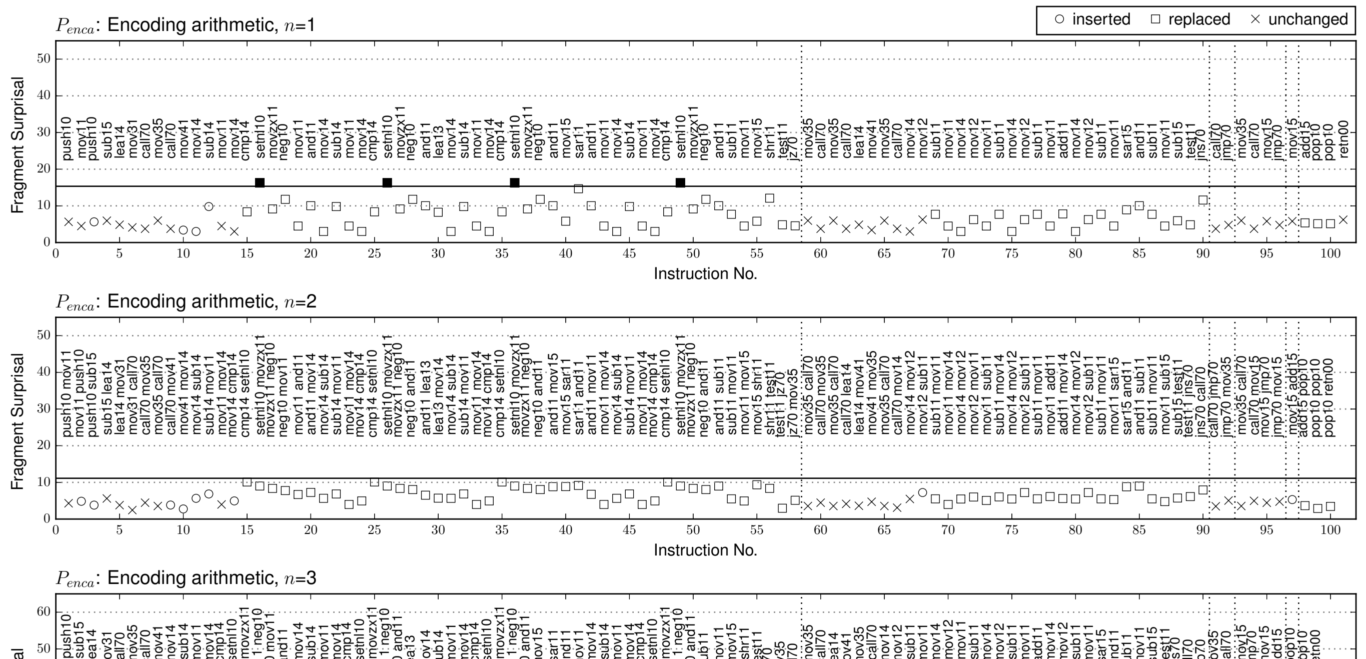


Figure 7: Distribution of surprisal of each fragment in P_{enca} (encoding arithmetic) for n=1, 2, 3.

```
if ((int )((unsigned long )(((((current_time -
    tmp) & - (current_time >= tmp)) + ((
    current_time - tmp) & - (current_time >= tmp
    ))) & (((current_time - tmp) & - (
    current_time >= tmp)) >> 63L)) - ((
    current_time - tmp) & - (current_time >= tmp
    ))) >> 63UL)) {
              :
}
```

Figure 8: Source-code obfuscation of an `if` statement in P_{enca}.

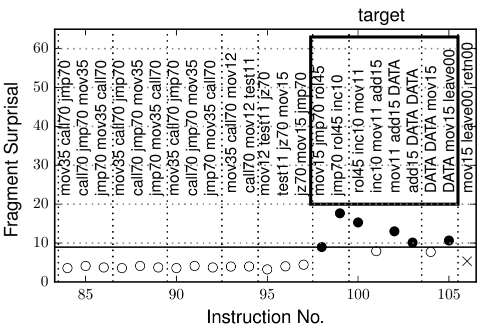
inserted by an obfuscation, and squares are fragments which have been replaced by an obfuscation.

Surprising obfuscations are filled circles or squares – these are true-positive detections. Surprising unobfuscated code fragments are bold-faced crosses – these are false-positive detections. Unsurprising obfuscations are unfilled circles or squares – these are false-negatives. Unsurprising unobfuscated fragments are non-bold crosses – these are true-negatives.

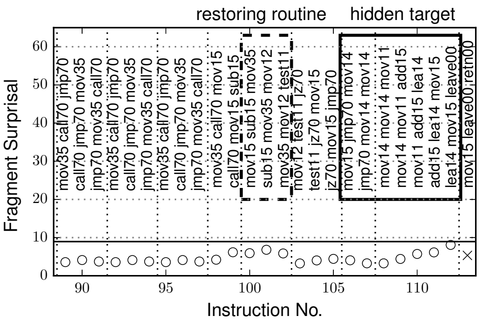
We observed no false-positives in P_o, P_{opaq}, or P_{vir}, for any of our models (n=1, n=2, n=3). We observed many false-negatives in P_{opaq} and P_{vir}. Apparently, most of the code inserted or modified by these obfuscations resembles some reasonably-common code sequences in our corpus.

From Figure 6, we can see that P_{opaq} has quite a few surprising n-grams. At instructions 98-102 of the top plot (for our 1-gram model), we see that the original code P_o had a sequence `mov15`, `jmp70`, and that the obfuscated code P_{opaq} inserts a series of instructions `rol45`, `inc10`, `mov11` immediately after that sequence. None of these insertions is very surprising in a 1-gram model, as their surprisals are all below our detection threshold. However the 2-gram `rol45-inc10` is very surprising, as are the 3-grams `mov15-jmp70-rol45`, `jmp70-rol45-inc10`, and `rol45-inc10-mov11`. We tentatively identify the fragments at 8-13 and 100-105 as unstealthy opaque predicates. We disassembled these sequences by hand, determining that they are indeed opaque predicates, and that they include some dummy code to prevent disassembly. A part of the dummy code is left undisassembled at 104-105 as `DATA-DATA`. This 2-gram is not surprising when it occurs in the text segment of a code in our corpus. From the bottom plot, we see that the `DATA-DATA` fragment is surprising if it immediately follows an `add15` instruction. We tentatively conclude that surprising anti-disassembly insertions will impede automated disassembly, but will aid pinpointing unless they are hidden by a subsequent pass through an obfuscating process.

In Figure 7, we note that P_{enca} contains four occurrences of the surprising 1-gram `setnl10` (set byte if not less). This instruction is being used in a series of bitwise-AND operations, in the obfuscated `if` statement of Figure 8. In the bottom plot (for n=3), we see four occurrences of the 3-gram `setnl10-movzx11-neg10`; this is apparently a signature of the data-encoding method as implemented in Tigress, allowing us to pinpoint its use. We note that this is not a very distinctive signature, for it is barely above the 8.95-bit threshold of our n=3 model at sensitivity θ=0.999. We noticed that compiling this code at a low level of optimization (`gcc -O1`) improves the stealthiness of this code by avoiding the `setnl` opcode. However, the level-1 optimizations of GCC also simplify the boolean expression in the guard of the `if` statement, so they decrease the strength of this obfuscation. In the following section, we show how to improve the stealth of this obfuscation without affecting its strength.



(a) before hiding process



(b) after hiding process

Figure 9: Example of hiding surprising fragments.

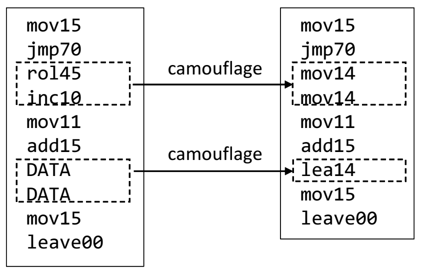


Figure 10: Hiding surprising fragments by instruction camouflage.

4.3 Example of hiding surprising fragments

In this section, we briefly discuss our feasibility study of the hiding process. We have not attempted to automate this process. Instead, we have manually identified surprising fragments, replaced them with unsurprising fragments of the same length, then repeated the process until there are no more surprising fragments. Somewhat to our surprise, we always made forward progress with our substitutions, that is, we never created any new surprising fragments by our code-modifications.

We implemented our hiding by an instruction camouflage [13], whereby dummy instructions are inserted in the static code, and are replaced dynamically at runtime with the original instructions. This greatly simplifies our hiding, as we have a free choice of dummy instructions. In particular, we can replace a surprising n-gram by any unsurprising continuation of its unsurprising (n-1)-gram prefix.

We illustrate this process for the surprising fragments at instructions 98-105 of P_{opaq}. Please see Figure 9. We replaced these instructions by a few "restoring routines" and a "hidden target". The restoring routines defined in [13] turn out to be unsurprising with respect to our models. One of the restoring routines is shown in dashed lines in Figure 9 (b). The others are scattered around the program. Each restoring routine consists of a few instructions, and could be just an inline insertion of a single `mov` instruction. In our experimentation, all of the restoring routines were unsurprising.

The hidden target would be unstealthy if it were merely a copy of the surprising instructions in the original code. Accordingly, we modified the suffix `rol45` of the first surprising fragment (substituting an unsurprising `mov14`), re-evaluated surprisals; then we scanned to the next surprising fragment, which ends with `inc10`. We replaced that suffix with the unsurprising `mov14`, and we also replaced the following 2-gram `DATA-DATA` with the 1-gram `lea14` (which is of the same length, in bytes). See Figure 10.

Our restoring routine therefore modified only a few instructions of the hidden target, with this modification being completed before the execution (inline) of the hidden target. We confirmed that the modified program P_{opaqh} had no surprising fragments, and that it ran correctly.

4.4 Discussion

4.4.1 Discussion of the pinpointing method

Our case study indicates that our pinpointing method is a promising way to detect some obfuscations, such as dummy code insertions and data encoding, because they introduce "surprising" sequences of instructions into the obfuscated code. Our pinpointing method is unlikely to detect other obfuscations, such as control-flow flattening and function virtualization, because they compile into "unsurprising" code such as `switch` statements and bytecode interpreters. We imagine that every obfuscating transform produces some characteristic signature whose features are, at least in principle, detectable. For example, a control-flow flattening obfuscation may produce code with surprisingly-long functions. A function virtualizer may produce an executable with a single, rather short, bytecode interpreter in its text segment, accompanied by a very long data segment. It is not reasonable to expect any pinpointing method to detect a very wide range of obfuscations. However we believe it is reasonable to expect any obfuscation to be stealthy with respect to a low-order n-gram model, because (as we have shown) surprising n-grams allow the pinpointing of the obfuscated code. In future work, we intend to investigate other features of obfuscated code which may allow it to be pinpointed, or at least to be detected.

In our case study, we considered only a single value of the detection threshold, $\theta = 0.999$. At this level of sensitivity, any fragment is considered to be surprising unless it occurs in our corpus at a frequency of at least 0.1%. In future work, we intend to characterize the sensitivity-specificity tradeoff in our obfuscation pinpointer, by producing "receiver operating characteristic" (ROC) curves for $n = 1, 2, 3$ on datasets containing obfuscations for which our detector is potent. For such datasets, there should be some correct detections (true positives) as well as many correct non-detections (true negatives). The rate of true positives is the "sensitivity" of the detection system, and will generally increase with θ. The rate of true negatives is the "specificity" of the detection system, and will generally decrease with θ. By convention in signal processing, the ROC curve is a plot of sensitivity as a function of the false-positive rate ($= 1-$ specificity). We also intend to experiment with higher-order models, and with the size of the corpus. We expect the detection performance (the area under the ROC curve) to increase with n, if the corpus is large enough to produce a stable n-gram model.

4.4.2 Discussion of the hiding method

In our case study, we illustrated a method for hiding surprising fragments that is based on the instruction-camouflage technique. Using this method, we were able to hide the surprising fragments we had pinpointed. This suggests that a hiding transform might be applied after the last stage of an obfuscating compiler, to ensure that its obfuscations are stealthy without affecting their strength. However we note that the instruction-camouflage technique has a recognizable feature: it is self-modifying code, which could be pinpointed either by a static points-to analysis, or by a dynamic analysis which identifies the modified instructions. A stealthier hiding transform could, we expect, be developed by adjusting the replacement-using-fundamental-instructions [14] obfuscation to ensure that it is creating only stealthy sequences.

5. RELATED WORK

Other publications describe alternative approaches to measuring the *strength* of an obfuscation. Examples include evaluating standard metrics of code complexity [9], queue-based mental simulation models [16], and controlled experiments involving human subjects [5]. There is also a study which proposes objective measure of obfuscation resilience against automated attacks using symbolic execution [4].

We are not aware of any prior quantitative measures of the the *stealth* of an obfuscation, aside from our recent measure of artificiality [12]. Our pinpointing metric is an extension of the artificiality metric. Both are based on an n-gram model. However artificiality is a property of an entire executable, whereas "surprise" (as defined in this paper) is a property of a short sequence of instructions.

6. CONCLUSION

In this paper, we proposed a pinpoint-hide method for improving the stealth of obfuscated code. In the pinpointing process, we scan short sequences of instructions in the obfuscated code, looking for surprising fragments – that is, for very unusual fragments which may draw the attention of an attacker to the obfuscated code. In the hiding process, we transform the surprising fragments into unsurprising ones, while preserving semantics. Because the obfuscated code transformed by our method consists only by unsurprising code fragments, it is more difficult for attackers to determine which (if any) fragments of an unknown executable have been obfuscated.

In the case study, we tested our pinpoint-hide method on a short program that had been transformed by five well-known obfuscation techniques. We found that our method can pinpoint surprising fragments, such as dummy code that does not fit in the context of the program, and also the instructions used in a complicated arithmetic expression that was introduced by an obfuscator. We confirmed that a method known as instruction camouflaging can be applied to the surprising fragments in the obfuscated code, resulting in a stealthier code which runs correctly and which has the same obfuscatory strength.

Our target program was intended to illustrate a prototypical cracking target. The callsite of this "Maginot protection function" could be easily modified, as soon as its functionality is identified. In future work, we intend to examine additional examples of security-sensitive code which has been obfuscated in a variety of ways, in order to determine whether our pinpointing method is sufficiently sensitive and selective to be of adversarial use. We will also determine whether a simple hiding technique, such as the one described in this paper, would improve the stealth of these security-sensitive codes.

We are intrigued by the possibility of applying a pinpointing method to dynamic traces of a code, given a test suite for its secure kernel or for some secure operation (such as license verification) which is a likely target for adversarial attack. Even if these routines are obfuscated with a function virtualizer, it would be possible to pinpoint any obfuscated operations with highly surprising traces.

Acknowledgment

This work was supported in part by Japan Society for the Promotion of Science (JSPS) under grant number 26330094.

7. REFERENCES

[1] The Cygwin project. http://www.cygwin.com/. (accessed: Sep. 2015).

[2] SRILM – the SRI language modeling toolkit. http://www.speech.sri.com/projects/srilm/. (accessed: Sep. 2015).

[3] B. Anckaert, B. De Sutter, D. Chanet, and K. De Bosschere. Steganography for executables and code transformation signatures. In *Proceedings of the 7th International Conference on Information Security and Cryptology*, ICISC'04, pages 425–439, Berlin, Heidelberg, 2005. Springer-Verlag.

[4] S. Banescu, M. Ochoa, and A. Pretschner. A framework for measuring software obfuscation resilience against automated attacks. In *Proc. IEEE/ACM 1st International Workshop on Software Protection (SPRO2015)*, pages 45–51, May 2015.

[5] 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*, pages 39–46, Alexandria, Virginia, USA, 2008.

[6] C. Collberg. The Tigress C diversifier/obfuscator. http://tigress.cs.arizona.edu/. (accessed: Sep. 2015).

[7] 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*, pages 319–328, Orlando, Florida, Dec. 2012.

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

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

[10] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages(POPL98)*, pages 184–196, San Diego, California, Jan. 1998.

[11] Hex-Rays. IDA support. https://www.hex-rays.com/products/ida/support/. (accessed: Sep. 2015).

[12] Y. Kanzaki, A. Monden, and C. Collberg. Code artificiality: A metric for the code stealth based on an n-gram model. In *Proc. IEEE/ACM 1st International Workshop on Software Protection (SPRO2015)*, pages 31–37, May 2015.

[13] 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*, pages 170–179, Dallas, USA, Nov. 2003.

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

[15] N. Mavrogiannopoulos, N. Kisserli, and B. Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, 2011.

[16] 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)*, pages 351–360, Sep. 2003.

[17] H. Tamada. Donquixote: A software obfuscation tool for Java programs. http://se-naist.jp/DonQuixote/. (accessed: Sep. 2015).

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

[19] P. Wayner. Mimic functions. *Cryptologia*, XVI(3):193–214, 1992.