An Experimental Survey of Data Obfuscation for Software Protection

Roberto Tiella, Mariano Ceccato
Fondazione Bruno Kessler, Trento, Italy
Email: {tiella,ceccato}@fbk.eu

***Abstract*—Program obfuscation is a largely adopted technique to protect code from malicious tampering, by making a program much more hard to understand. In fact, before altering it, a potential attacker has first to understand the program and then to elaborate an attack strategy.**

While the results of the research on control flow obfuscation are already largely adopted, approaches and tools for obfuscating and protecting program data are less popular.

This paper presents a survey of the existing approaches to obfuscate program data. Relevant state-of-the-art approaches are presented and compared with the purpose of highlighting applicability conditions, strengths and weaknesses. We also implemented selected data obfuscation approaches, to empirically measure the obfuscation cost in terms of performance degradation.

I. Introduction

Programs often enforce usage conditions, e.g., license check, that could be broken in case the code is tampered with by malicious users (man-at-the-end attack model [1]). Among the possible protection strategies that can be applied to a program to limit malicious program understanding, data obfuscation aims at hiding data values by changing the statements where variables are defined and used.

Theoretical results [2] limit the general applicability of obfuscation, because a determined attacker might still understand obfuscated code, provided that she/he spends enough time and resources in this task. However, obfuscation can be still considered a winning protection strategy when the cost of attacking obfuscated code (e.g., attacker time) is more expensive than the benefit of the successful attack (e.g., the cost of the license).

Despite many approaches for obfuscating the program *control flow* have been defined [3] and are quite largely adopted in practice, fewer approaches directly target the obfuscation of program *data*. For example, basic program obfuscation is already integrated in software development tools to protect smart phone apps against malicious reverse engineering [4], but data obfuscation is still not largely adopted.

However, even if the control flow would be extremely difficult to understand, if values are stored in clear they could be easily attacked, without the need of fully break control flow obfuscation. In fact, clear data could be read from the binary program before execution, or at runtime from the program memory. For this reason, *data* obfuscation should complement *control* obfuscation in automated tools that supports software development.

However, data obfuscation transformations vary a lot in terms of effectiveness, applicability conditions and performance overhead. These features should be clear to a developer who should decide which obfuscation better fits her/his application requirements.

In this paper, we present a survey of data obfuscation approaches to protect C programs against man-at-the-end attacks. We identified the relevant approaches in literature. We compared them in terms of the obfuscation features that they offer, their limitations and the applicability pre/post-conditions.

Moreover, we implemented a representative subset of these obfuscations, strictly following the algorithm description by original authors. We applied the implemented obfuscations to a set of C applications, to study obfuscation cost in terms of execution time degradation and memory overhead.

II. Data Obfuscation Survey

Figure 1 shows the classification for data obfuscations derived by the one presented by Collberg et al. [5] when considering only transformations for `C` (i.e., removing obfuscations specific for object-oriented code).

The first classification level distinguishes transformations in: (a) *Storage & Encoding*, i.e., transformations that change how (scalar) data are represented and stored in memory; (b) *Aggregation*, i.e., transformations that alter how data, both scalar variables and arrays, are aggregated; and, (c) *Ordering*: i.e., transformations that permute items in data structures as, for example, changing the order of items in an array of integers. The relevant features and limitations of obfuscations in this survey are described in the rest of this section, and summarized in Table I.

A. Storage and Encoding

As described by Collberg et al. [5], obfuscating storage transformations attempt to choose non conventional memory layout for dynamic as well as static data. Splitting a single 32 bits integer variable in four 8 bits variables is an example of storage transformation. Similarly, encoding transformations attempt to choose unnatural encodings for common data types. Encoding a boolean value as an integer value where any even number represents *True* and any odd number represents *False* is an example of encoding transformation for the C programming language. Storage and encoding transformations

often go hand-in-hand, but they can sometimes be used in isolation.

We present in more details four storage/encoding obfuscations, namely XOR masking, Residue Number Coding, Merge Scalar Variables and Convert Static Data to Procedure.

XOR masking. A simple but quite frequently [6] used encoding function involves the bitwise XOR operator. If $\oplus$ denotes the XOR bitwise operator, the encoding function $e(\cdot)$ is defined as follows:

$$e(x) = x \oplus p$$

where p is an integer constant. From the property of the XOR operator that $(x \oplus p) \oplus p = x$, it follows that the decoding function $d = e$. This encoding is usually called *XOR masking*.

Figure 2 contains snippets of code before and after having applied the XOR masking (with $p = 12$). The values of a, b and x are stored encoded in memory, and they are decoded as the clear values are needed (in the computation of the sum and in the *printf* statement). XOR masking (and similar encodings) are known to be employed by practitioners, and by malware programmers in particular [7].

This obfuscation is not expected to cause major overhead in terms of memory increase and execution time degradation. As preparation for this obfuscation, reliable points-to information should be available, to know when two variables are alias. In fact, if a variable is encoded, all its alias should be consistently considered obfuscated.

Static points-to analysis specifies when two variables *may* be aliases. There are programs for which precise points-to information can not be computed statically so it would be impossible to know if variables are or are not aliases (precise pointer analysis is, in general, undecidable [8]). However, to apply some data obfuscation transformations, precise points-to information is required and the programmer should inspect the output of static pointer analysis and refine it, e.g., by manually adding code annotations with precise points-to and alias information.

1) Storage and Encoding
   a) Change Encoding
      - XOR masking
      - Residue number coding
      - Linear coding
   b) Split variables
   c) Convert static data to procedure
   d) Change variable lifetimes
2) Aggregation
   a) Merge scalar variables
   b) Restructure arrays: split, merge, fold, flatten arrays
3) Ordering
   a) Reorder arrays

Fig. 1. Data obfuscation taxonomy (adapted from [5]).

```
int a = 5;              int a = 9; // 9 = 5^12
int b = 8;              int b = 4; // 4 = 8^12
int x = a+b;     ⇒      int x = ((a^12)+(b^12))^12;
...                     ...
printf("%d\n",x);       printf("%d\n",x^12);
```

Fig. 2. Example of XOR masking.

Linear Coding. *Tigress*[1] is an obfuscation tool developed by Collberg [9] featuring, among other obfuscation techniques, data obfuscation. *Tigress* support to data obfuscation comprises integer data encoding by means of integer affine transformation, i.e. transformations having the following form:

$$e(x) \rightarrow ax + b$$

with $a, b \in \mathbb{Z}$.

Residue Number Coding. A more advanced case of encoding is the one that does not require to decode back encoded values to operate them. This is the case of *homomorphic* encodings. Formally speaking, an encoding $e(\cdot)$ is an *homomorphism* if it satisfies, for all x_1, x_2 in the set X on which $e(\cdot)$ is defined, the following condition:

$$e(x_1 + x_2) = e(x_1) +' e(x_2). \quad (1)$$

When a homomorphic encoding function $e(\cdot)$ is used, source code obfuscating transformations can be simplified leveraging Eq. 1, holding:

$$e(d(a) + d(b)) = e(d(a)) +' e(d(b)) = a +' b.$$

The simplification avoids the need to decoding encoded values to perform the operation.

Residue Number Coding (RNC)[10] is an encoding on integers which is homomorphic for both the sum (and thus the subtraction) and the product. Having chosen $m_1, m_2, ..., m_u \in \mathbb{Z}$ so that $gcd(m_i, m_j) = 1$ if $i \neq j$, RNC encodes $x \in [0, n-1]$, where $n = m_1 \cdot m_2 \cdot ... \cdot m_u$, in the following way:

$$e(x) = ([x]_{m_1}, [x]_{m_2}, ..., [x]_{m_u}).$$

The decoding function d is defined such that:

$$d([y]_{m_1}, [y]_{m_2}, ..., [y]_{m_u}) = [y],$$

where $[y]$ exists and it is unique by the "Chinese Remainder Theorem" and can be computed using Euclid's extended algorithm to compute the gcd [11]. Operations in the encoded domain are defined "per component":

$$([x_1]_{m_1}, [x_2]_{m_2}, ...) + ([y_1]_{m_1}, [y_2]_{m_2}, ...) = ([x_1 + y_1]_{m_1}, [x_2 + y_2]_{m_2}, ...)$$

$$([x_1]_{m_1}, [x_2]_{m_2}, ...) * ([y_1]_{m_1}, [y_2]_{m_2}, ...) = ([x_1 * y_1]_{m_1}, [x_2 * y_2]_{m_2}, ...)$$

Figure 3 shows an example of encoding integer variables x and y using RNC. Each original value is split in two encoded values, because two constant numbers are used as encoding base, they are are $m_1 = 31$ and $m_2 = 37$. To increase

[1] http://tigress.cs.arizona.edu/

obscurity of values, random representatives[2] are chosen for constant values.

```
int x = 12;
int y = 7;

int z = x+y;
int w = x*y;

printf(
  "z=%d, w=%d\n",
  z, w);
```

⇒

```
// 1965 % 31 = 12
int x1 = 1965;
// 1973 % 37 = 12
int x2 = 1973;

// 1433 % 31 = 7
int y1 = 1433;
// 2634 % 37 = 7
int y2 = 2634;

int z1 = x1+y1;
int z2 = x2+y2;

int w1 = x1*y1;
int w2 = x2*y2;

printf("z=%d, w=%d\n",
       d(z1,z2),d(w1,w2));
```

Fig. 3. Example of encoding integer variables using Residue Number Coding.

Due to the intense computation on encoding/decoding, this obfuscation is expected to convey sensible execution time overhead. Moreover, also some memory overhead is expected due to the fact that a clear variable is represented by two new variables once obfuscated. Before applying this transformation, one needs to check the value range of the variables to obfuscate, to make sure that they can fit in the (reduced) encoded interval. Developers have to provide this information.

Split Variables. Boolean variables and other variables of restricted range can be split into two or more variables [12], and operations among split variables are supported by look-up tables. The complexity of operations and the size of the look-up table depend on the range of variables. More formally, if a variable x of type T is mapped into n variables of type U by means:

- The splitting (encoding) function $e : T \rightarrow U^n$;
- Its inverse $d : U^n \rightarrow T$;
- A set of (usually tabulated) operations on U^n so that $e : T \rightarrow U^n$ preserves operations.

For example boolean values can be split according to the following scheme. Let's consider a 0/1 representation for truth values, i.e., $x \in \{0, 1\}$, $n = 2$ and $U = \{0, 1\}$. Define the encoding function $e(x)$ so that

$$e(x) \in \begin{cases} \{(0,0),(1,1)\} \text{ if } x = 0 \\ \{(1,0),(0,1)\} \text{ if } x = 1 \end{cases}$$

The corresponding decoding function is defined as $d(y_1, y_2) = mod(y_1 + y_2, 2)$. It can be easily proven that the function $d(y_1, y_2)$ is the inverse of the function $e(\cdot)$. For each operation among split values, the approach needs to define a static $U^2 \times U^2$ table (in the example it is a table 4x4). The '&&' operator can be defined in the encoded domain, namely U^2, using the table $\texttt{AND}(i, j)$: $\texttt{e(x \&\& y)} = (\lfloor z/2 \rfloor, mod(z, 2))$ where $z = \texttt{AND}(2e_1(x)+e_2(x), 2e_1(y)+e_2(y))$. Similar tables can be produced for '||' and '!' operators.

Similarly to residue number coding, also this obfuscation supports operations in the obfuscated domain, but limited to those for which a tabulated function is available. Split Variables is expected to bring execution time and memory overhead. It is critical to identify the vale range of variables to obfuscate to prepare the code for obfuscation.

Convert Static Data to Procedure. This is an elaborate encoding function proposed for hiding static data such as character strings. The idea is to transform a chunk of static data into some code that once invoked produces the original data [12]. One of the possible implementations is based on Mealy machines as illustrated in Collberg and Nagra's book [13]. A Mealy machine is defined by a set of states $S = \{s_1, ..., s_n\}$, an input alphabet $X = \{x_1, x_2, ..., x_p\}$, an output alphabet $Y = \{y_1, ..., y_q\}$ and two characterizing functions f_y (the output function) and f_s (the transition function):

$$y_k = f_y(x_k, s_k)$$

$$s_{k+1} = f_s(x_k, s_k)$$

A Mealy machine encodes a (partial) function from words on the (encoded) input alphabet X to words on the (decoded) output alphabet Y. Proper invocations to the procedure that implements the Mealy machine can replace usages of static data.

This obfuscation is expected to cause a medium runtime execution overhead, proportional to how many times the Mealy machine is invoked to generate the obfuscated constant string. To apply this obfuscation, the code needs to be prepared so that strings are clearly static constants. Moreover, the obfuscated code needs to be post-processed to *free* the memory allocated to the string, to void memory leaks.

B. Aggregation Transformations

Aggregation transformations change how data, both scalar variables and arrays, are arranged in memory.

Merge Scalar Variables. Two or more scalar variables V_1 ... V_n can be merged into one variable W, provided the combined ranges of V_1 ...V_n fit within the precision of W:

$$W = 2^{k_1}V_1 + ... + 2^{k_n}V_n$$

Where k_i is the position in W where variable V_i starts [12]. We assume $n = 2$ in the previous equation in what follow to ease the explanation, i.e.,

$$W = 2^k V_1 + V_2$$

Operations on the original variables can be mapped to operations on the merging variable

$$\begin{array}{lll} V_1 = V_1 + d & \Rightarrow & W = W + 2^k d \\ V_1 = V_1 * d & \Rightarrow & W = W + (d-1)(W\&(2^k - 1)), \end{array}$$

where '&' is the bitwise AND operator.

[2]The congruence class $[x]$ is represented in the program by a value $\hat{x}$. Instead of having straightforward $\hat{x} = x \mod m_1$, we select a value more difficult to guess, $\hat{x} = (x \mod m_1) + r * m_1$, where r is a random value.

```
                              #define MASK
                                ((((long)1) << 32)-1)

                              // w stores (x,y)
int x,y;                      long w;
int d,e;                      int d,e;

x = 31;                       // w = (x,y)
y = 24;                       w = (((long)31) << 32)
d = 12;                           + 24;
e = 5;                   ⇒
                              d = 12;
x = x + d;                    e = 5;
y = y * e;
                              // w = (x+e,y)
printf("x=%d, y=%d\n",        w = w + ((long)d << 32);
       x,y);                  // w = (x+e,y*d)
                              w = w + (e-1)*(w & MASK);

                              printf("x=%d, y=%d\n",
                                     w>>32,w&MASK);
```

Fig. 4. Merge scalar variables example.

Figure 4 shows an example of this obfuscation where two int variables (holding positive values) are fitted into a long variable[3]. When clear values are needed, decoding is required using shift operators and masks.

Code obfuscated with Merge Scalar Variables should have minor execution and memory overhead. To apply this transformation, reliable points-to information is required and the range of program variables should be provided, in particular to understand the sign, to be used consistently in shift operations.

Restructure Arrays. Arrays can be reshaped to harden the task of statically determining their content [5]. Confusing the expressions used as indexes and, consequently, obfuscate how elements are accessed is another application of these techniques. Transformations that can be applied to arrays are:

a) Splitting: A single array of integers is halved in two arrays, putting items with even index in the first array and items with odd index in the second one.

b) Merging: The opposite operation of splitting. Two arrays are merged so that the elements of the first array are put in even-indexed positions while elements from the second array are put in the odd ones.

c) Folding: Array folding takes a unidimensional array and converts it in a two-dimensional array splitting it in two rows. Note that the memory content is not changed, only index expressions to access specific array elements are changed.

d) Flattening: Array flattening takes a two-dimensional array and convert it in a unidimensional array by joining its rows. Note that, as in the case of folding, the memory content is not changed.

Restructure array is considered a quite cheap obfuscation, as it should bring very low overhead. Only the *splitting* variant could have a medium execution overhead due to the decision on which array to reference in the obfuscated version.

[3]In the example we assume a 64 bit architecture.

For *splitting* and *merging*, a preparation step required to make sure that array elements are accessed by index, bulk operations such as `memcpy` are not supported. *Folding* and *flattening* do not change memory content or layout, but their aim is to make it harder to understand which element is indexed. Thus, the use of pointers does not impede the code to be obfuscated, but it makes the obfuscation useless.

C. Ordering Transformations

Ordering transformations permute items in data structures.

Reorder Arrays: One very naive technique occasionally used by practitioners to try hiding strings embedded in a program from straightforward searches, is to reverse the order of elements in strings. In general any index permutation $f(i)$, can be applied to array/string elements to achieve a more resilience transformation. Function $f(i)$ can be implemented by means of a look-up vector or an expression. Zhu and Thomborson[14] propose to use the homomorphic function $f(i) = i * m \ (mod \ n)$ which is a isomorphism if m and n are co-prime. As for other types of transformations on strings, a per-character `putchar` must replace the original `printf` function. In general the original string/array must be reconstructed in memory if it has to be passed to any external function, such as `strcmp`.

The execution time overhead of Reorder Arrays is quite negligible, but the memory overhead, to store the indexes permutation, depends on the array length and on the index size. The code should be prepared for this obfuscation to be effective, and strings should be turned as static constants. No pointer arithmetic should be used on string obfuscated in this way.

III. Obfuscation Configuration: Data Distance

As underlined in Sections I and II data obfuscation transformations cannot be applied for free. Gain in obscurity and loss in runtime performance must be traded and an obfuscation strategy must be devised. The most conservative strategy to

```
1  //safe vars
2  int a,b,c,d;
3
4  //sensitive vars
5  int x,y,z;
6
7  a = 2;
8  b = a + 3;
9  x = a + b;
10 c = 12;
11 y = b + 1;
12 z = c + y;
13 d = 2 * c;
14 z = x;
```

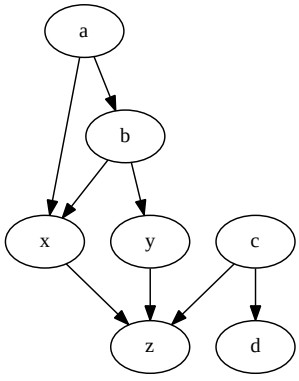


Fig. 5. An example of C snippet and the related DPG. x, y and z are sensitive variables while a, b, c and d are not.

prevent code tampering is *obfuscate all*, i.e., to apply obfuscation to all the program variables. However, not all the variables in a program are expected to be security critical, e.g., variables

Technique	Apply to types	Execution time overhead	Memory overhead	Preparation	Homomorphic operations	Manual post-process	Preconditions
XOR masking [6]	Integral	low	none	Points-to info			
Linear coding [9]	Integral	low	none	Identify range, Points-to info			
Residue number coding [10]	Integral	high	#variables x 2	Identify range, Points-to info	sum (+), product (∗)		
Split Variables [12]	Small range integer (typically boolean)	medium	#variables x 2 + 1 matrix per operation (4 range ^ 2)	Identify range, Points-to info	for operations with matrix		
Convert static data to procedure [13]	Static array	low/medium	buffer length x 3 + malloc size	prepare static string		need to call `free(...)` (to prevent memory leaks)	
Merge scalar variables [12]	Integral	medium	none	Points-to info, Identify sign	sum (+), product (∗) with clear values		
Restructure array (split) [5]	Statically allocated array (no malloc)	low/medium	none	array access by index			no pointer arithmetic
Restructure array (merge) [5]	Statically allocated array (no malloc)	low	none	array access by index			no pointer arithmetic
Restructure array (fold) [5]	Statically allocated array (no malloc)	low	none				pointer arithmetic work but prevent obfuscation of index access
Restructure array (Flatten) [5]	Statically allocated array (no malloc)	low	none				pointer arithmetic work but prevent obfuscation of index access
Reorder array [14]	Statically allocated array (no malloc)	negligible	length of vector x index size	prepare static string			no pointer arithmetic

TABLE I
FEATURE VALUES FOR EACH SPECIFIC DATA OBFUSCATION TECHNIQUE.

related to the GUI might not represent a security threat in case of tampering. Thus, such an aggressive approach could cause unmotivated and unacceptable performance degradation. The opposite strategy is *obfuscate just sensitive*. It consists of obfuscating only the particular program variables that are security sensitive and are prone to attacks, e.g., those variables in strategy games that store gold and energy values. However, other variables that are not intrinsically sensitive could be somehow related to sensitive variables. Related variables could leak important information that could be potentially used by the attacker to guess or tamper with the value of a sensitive variable.

With respect to the example of Figure 5, we see that at line 11 variable `b` is assigned to variable `y`. If an attacker knows or tampers with the (clear) value of `b` before line 11 , the value of `y` is known or tampered with when the execution reaches line 11, even if `y` is obfuscated. Thus, the strategy *obfuscate just sensitive* is also suboptimal, in fact related variables should also be considered for obfuscation. Intuitively, values that participate to the definition of a sensitive value should be also obfuscated, so they should be variables that are defined using sensitive values. To capture this intuition, we propose to consider neighbourhoods of certain size around sensitive variables in the *Data Proximity Graph* as presented in the next section. An intuition of this approach was sketched in a master thesis [15] and fully elaborated in this paper.

A. Data Proximity Graph and Data Distance

Given two variables v_1 and v_2, we say v_1 is *proximal* to v_2, written as $v_1 \rightarrow v_2$, if v_1 is used to define the value of

v_2. In the example of Figure 5, because of the assignment on statement 11 `y=b+1`, variable `b` is proximal to variable `y`, i.e., $b \rightarrow y$.

Given a program, the Data Proximity Graph (DPG) for the program is a directed graph where nodes are the variables $\{v_i\}$ and there exists an edge between nodes v_1 and v_2 iff $v_1 \rightarrow v_2$, namely if v_1 is proximal to v_2. The complete DPG for the example is reported in Figure 5.

Informally, on the DPG, the distance between two nodes is the number of assignments that we have to traverse to use the first variable in an assignment of the second one. Using the DPG it is possible to define a distance, called *data distance* between variables in a program. More formally the distance between two variables v_1 and v_2 is defined as

$$d(v_1, v_2) = min\{|P| : P = path_{DPG}(v_1, v_2)\} \qquad (2)$$

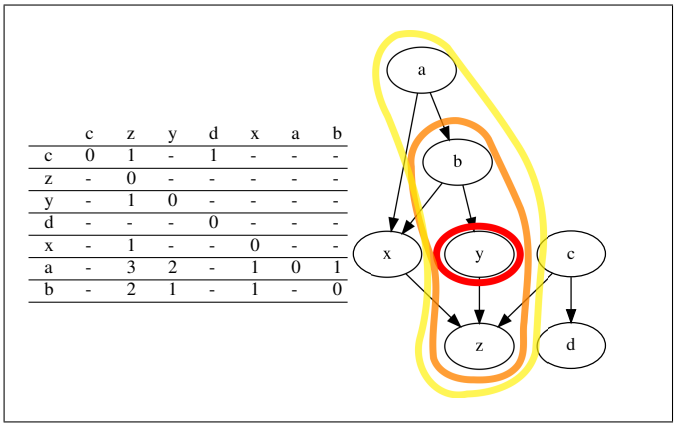
In our running example, there is an edge from `z` to `c`. For this reason data distance $D(z, c)$ is 1. From this graph we can see that the distance between `z` and `b` is 2, the distance between `b` and `a` is 1 and the distance between `z` and `a` is 2. In Figure 6 the table shows the distances among variables for example in Figure 5.

B. Neighbourhood of a Variable

Given a (sensitive) variable v, the *neighbourhood of v with radius r*, $N_r(v)$, can be defined as:

$$N_r(v) = \{v' | d(v, v') \leq r \vee d(v', v) \leq r\}.$$

In other words, $N_r(v)$ is the set of variables that reaches v or are reached by v in r assignments. Figure 6 shows neighbourhoods of y with radius 0 (in red), 1 (in orange) and 2 (in yellow). Remaining variables cannot reach y or cannot be reached by y. Given a sensitive variable v, its neighbourhood will contain more and more variables as the radius increases, defining a growing sequence of subsets of variables that could be obfuscated to increase the level of data obscurity of the code v belongs to.



	c	z	y	d	x	a	b
c	0	1	-	1	-	-	-
z	-	0	-	-	-	-	-
y	-	1	0	-	-	-	-
d	-	-	-	0	-	-	-
x	-	1	-	-	0	-	-
a	-	3	2	-	1	0	1
b	-	2	1	-	1	-	0

Fig. 6. Variables data distance matrix and neighbourhoods of a sensitive variable

C. DPG Implementation

The computation of the Data Proximity Graph was implemented as an extension of GrammaTech CodeSurfer [16]. CodeSurfer is a tool able to analyse C/C++ programs. It generates a set of representations, including Abstract Syntax Trees, Control Flow Graphs and the System Dependence Graph. CodeSufer offers C and Scheme API to interact with these representations.

The computation of Data Proximity Graph has been implemented as a batch script using the Scheme API offered by CodeSufer. The DPG is then passed to a graph visiting algorithm that extracts the neighbourhood of a given radius for a given variable.

IV. Comparison Settings

Data obfuscation is supposed to make programs more difficult to understand and, consequently, to tamper with. However, security comes at some cost. In the present study we are quantifying this cost, with the following research questions:

RQ$_1$: What is the *memory overhead* of data obfuscation?

RQ$_2$: What is the *execution time overhead* of data obfuscation?

A. Metrics

The answer these research questions we measure these metrics

- **NOBV**. The obscurity of the code (i.e., the level of protection) is represented by the number of the program variables that are subject to data obfuscation. After identifying a sensitive variable, we gradually increase the radius of the neighbourhood in the DPG to include more and more variables in the set of variables to obfuscate.
- **MEM**. The size of (static) memory allocated to program variables. The memory size is computed using a Scheme script written for the CodeSufer. The script collects all variable declarations, for each declaration retrieves the allocated size, and finally sums up the collected values to obtain MEM.
- **ETIM**. The execution time (ETIM) is measured by means of the `time` utility. In Unix systems, the time used by a process to execute is accounted as *system time* and *user time*. To avoid that the constant setup time required to start a process dominates the actual execution time, we added an artificial loop that executes the main program several times. Moreover, we added an artificial dependency to avoid the compiler to remove this loop.
- **IENC & IDEC**. The number of encoding/decoding function invocations (IENC/IDEC) is computed by instrumenting the obfuscated code. Every time an encoding or decoding operator is applied a related counter is incremented. Measures are averaged across several execution scenarios.

IENC/IDEC and ETIM were collected in separated runs, to avoid that counting encoding and decoding could influence the measurement of the execution time.

B. Treatments

The treatment is represented by obfuscated code, while the control group is represented by the original clear code (i.e., when no obfuscation is deployed). In this experiment we consider a selection of relevant state-of-the-art data obfuscation transformation among those presented in Section II:

1) XOR masking;
2) Variable Merging; and
3) Residue Number Coding (RNC).

We implemented these transformations as source-to-source transformations for C code, strictly following the algorithm presented by the cited papers. Obfuscation transformations are implemented using the TXL programming language [17]. TXL is a rule-based functional language specifically designed for parsing code and rewriting the parse trees.

C. Subject Applications

Table II lists the set of applications we used in the experiment, together with their size measured as non-blank lines of code. *License* is a routine devoted to check the validity of a license number to activate a software component.

The security sensitive variable is the one that holds the difference in days between emission date and current date. In fact, an attacker might tamper with this value to make an expired license last longer. An attacker might (i) add a constant value to the current date, (ii) subtract a constant value from the difference, or (iii) add a constant value to the date extracted from the license number. All these examples are instances of attacks based on data tampering that could be mitigated using data obfuscation.

Of course many other attacks are possible based on other strategies, such as tampering with the code to skip license validation, altering the system date, or tamper with the system library that fetches the current date. All these attacks are out of the scope of data obfuscation. Different protections are effective against these attacks, such as code obfuscation or remote attestation.

Arithmetic is a program that proposes simple arithmetic problems to the user. *Lottery* is an utility that draws samples from the random generator provided by a smartcard (a simulator for the smartcard is included). *Opchain* is randomly generated synthetic computational intensive program consisting in a sequence of assignment statements involving integer variables and the four operators '+','-','*' and '/'.

All the subject applications come with a set of test cases, to be used to verify their correct behaviour when obfuscation is applied.

Application	LOCS
arithmetic	265
license	169
lottery	165
opchain	90

TABLE II
SUBJECT APPLICATIONS USED IN THE EXPERIMENT.

D. Experimental Procedure

To study the impact of obfuscation, a set of execution scenarios have been defined for the case study. They correspond to the test cases available in the applications. For example, for license there are 100 scenarios, corresponding to licenses emitted on different dates. Half (50/100) of the licenses are valid, the reset (50/100) of them are expired.

As the execution of the original license check program was quite fast, we artificially modified the program by introducing an iteration of $50 \cdot 10^6$ executions of the *main* function. This way, we are able to have the program execution lasts for tens of seconds, improving the precision for the measurement of ETIM. Table III reports how many artificial iterations we added to obtain a similar result for all the subject applications.

Application	Iterations
arithmetic	5,000,000
license	50,000,000
lottery	100,000
opchain	500,000,000

TABLE III
ARTIFICIAL ITERATIONS OF THE ORIGINAL MAIN FUNCTION.

Given a sensitive variable v, the radius r of its neighbourhood can be used as parameter to numerically describe the obfuscation strategy and consequently the strength and expected overhead for data obfuscation. We, thus, define multiple obfuscation configurations, corresponding to a different set of variables to be obfuscated as the considered radius r becomes bigger and bigger. The number of obfuscated variables is measured as NOBV.

Configurations are used to apply each obfuscation technique. Eventually, we measure MEM on the obfuscated code, and we run the obfuscated programs to collect ETIM and IENC/IDEC values. To make sure that data obfuscation preserves the original semantics, on all the scenarios output of the obfuscated code is compared to the output of clear code.

The experiment has been conducted on a Desktop with Intel Xeon 3.3 GHz CPU (4 cores), 16 GB of memory, running Red Hat 6.5 64 bit.

V. COMPARISON RESULTS

This section reports the results of our experimental comparison of selected obfuscations.

A. Memory Overhead

Table IV lists how size (expressed in number of bytes) of required memory is affected by data obfuscation techniques. XOR masking does not affect memory, because original and obfuscated variables take the exact same amount of bytes in memory.

Code obfuscated with Merge Variables requires more memory than the clear code. In fact, this obfuscation packs two char or int variables in a single long variable (packing of long variables is not supported). In case two char variables (1 byte each, 2 bytes in total) are merged into a long variable

		Size (% increase)		
Type	Original	XOR	Merge Vars	RNC
char	1	1 (0%)	8/2 (300 %)	16 (1500%)
int	4	4 (0%)	8/2 (0 %)	16 (300%)
long	8	8 (0%)	- (-)	16 (100%)

TABLE IV
SIZE OF ORIGINAL AND TRANSFORMED DATA BY OBFUSCATION TECHNIQUE.

(8 bytes), the memory expansion is from 2 bytes to 8 bytes, corresponding to +300% of memory (in the table, the notation 8/2 means that each variable occupies half of the final long variable). No additional memory, instead, is required when two int variables (two times 4 bytes) are merged in a long variable (8 bytes).

In our implementation, code obfuscated with RNC stores each encoded value in an array of size 2, that occupies 16 bytes. Memory overhead is of 1,500% for char variables, 300% for int variables and 100% for long variables.

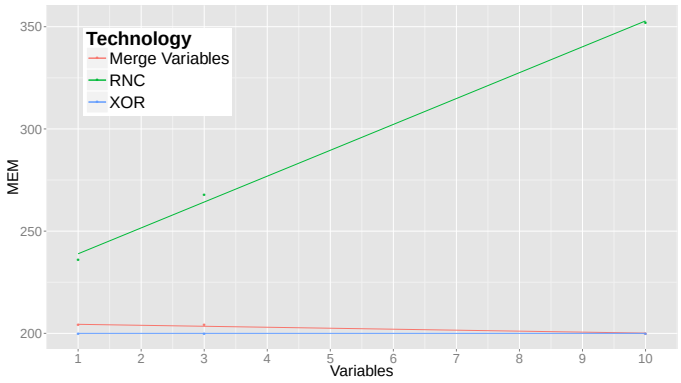


Fig. 7. Memory overhead (MEM) in *license* for obfuscated code with increasing number of obfuscated variables (NOBV).

Figure 7 show the experimental results of memory overhead for *license* when more and more variables are subject to data obfuscation. The memory overhead show a liner trend only when code is obfuscated with RNC.

To understand if the linear trend observed in the graph is significant, we use the *Pearson correlation* test. This test computes the correlation coefficient ρ, a measure of the strength of the linear relationship between two variables. It ranges from -1 to +1, where the extremes indicate perfect (positive or negative) correlation and 0 means no correlation. Statistical significance is assumed when this test reports a p-value is $<$0.05 (we assume significance at a 95% confidence level, α=0.05), significant cases will be highlighted in boldface.

Table V shows Pearson's correlation between the number of obfuscated variables (NOBV) and memory (MEM) in actual programs. The only statistical significant case is for RNC, where the average additional memory required ranges from 12 bytes to 16.90 bytes for each obfuscated variable.

	XOR			Merge Variables			RNC		
SUT	ρ	p-val	m	ρ	p-val	m	ρ	p-val	m
arithmetic	NA	NA	0.00	0.81	0.19	1.33	**1.00**	**0.01**	**16.9**
license	NA	NA	0.00	-0.98	0.14	-0.48	**1.00**	**0.04**	**12.7**
lottery	NA	NA	0.00	-0.45	0.55	-0.80	**1.00**	**0.01**	**13.2**
opchain	NA	NA	0.00	-0.51	0.30	-0.14	**1.00**	**0.01**	**12.0**

TABLE V
PEARSON'S CORRELATION BETWEEN THE NUMBER OF OBFUSCATED VARIABLES (NOBV) AND MEMORY (MEM).

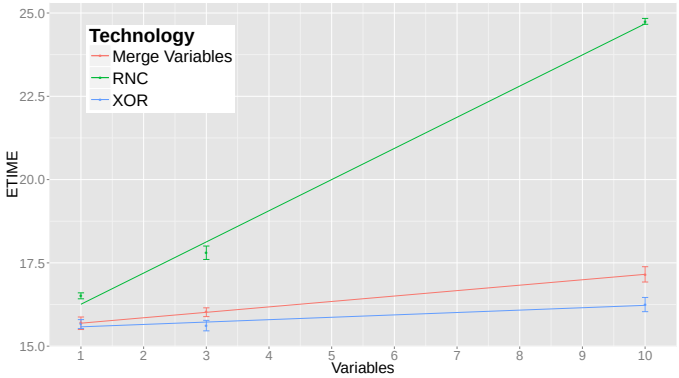


Fig. 8. Runtime overhead (ETIME) in *license* for obfuscated code with increasing number of obfuscated variables (NOBV).

B. *Runtime Overhead*

Figure 8 shows the runtime overhead for *license* when more and more variables are subject to code obfuscation. The runtime overhead seems to follow a liner trend in the number of obfuscated variables. A small and similar overhead is recorded for XOR masking and Variable Merge, while Residue Number Coding looks more time expensive, but still linear.

Table VI reports the results of the Pearson correlation between the number of obfuscated variables (NOBV) and runtime overhead (ETIME), the p-value and the slope m of the interpolating line. As already observed in Figure 8, when code is obfuscated with XOR Masking the execution slow down is quite limited. A significant performance degradation could be observed only on *opchain*, and it consists of 0.34 additional seconds per each additional obfuscated variable (when the program is iterated 50,000,000 times, see Table III).

Code obfuscated with Merge Variables is affected by higher runtime overhead, that is statistically significant in two cases, they are *license* and *op-chain*. The slowdown is respectively of 0.16 seconds and 0.82 seconds per each obfuscated variable (when the code is iterated respectively 50 and 500 millions time).

Data obfuscation based on Residue Number Coding is the most time consuming. While in *lottery* no significance slowdown is observed, on *arithmetic*, *license* and *op-chain* the slowdown is significant and can be quantified respectively in 0.81, 0.94 seconds and 8.79 seconds per obfuscated variable.

To study the reason for this performance degradation, we now consider the number of encoding/decoding operations

SUT	XOR			Merge Variables			RNC		
	ρ	p-val	m	ρ	p-val	m	ρ	p-val	m
arithmetic	-0.37	0.63	-0.06	-0.82	0.18	-0.08	**0.99**	**0.01**	**0.81**
license	0.96	0.18	0.07	**1.00**	**0.01**	**0.16**	**1.00**	**0.04**	**0.94**
lottery	-0.77	0.23	-0.01	-0.39	0.61	-0.01	-0.01	0.99	-0.01
opchain	**0.98**	**0.01**	**0.34**	**0.99**	**0.01**	**0.82**	**0.98**	**0.01**	**8.79**

TABLE VI
PEARSON CORRELATION BETWEEN THE NUMBER OF OBFUSCATED VARIABLES AND EXECUTION TIME.

SUT	XOR			Merge Variables			RNC		
	ρ	p-val	m	ρ	p-val	m	ρ	p-val	m
arithmetic	**1.00**	**0.00**	**101**	**1.00**	**0.00**	**101**	**0.99**	**0.01**	**66**
license	1.00	0.05	162	1.00	0.05	162	0.91	0.27	48
lottery	0.86	0.14	1347	0.86	0.14	1347	0.90	0.10	898
opchain	**0.99**	**0.00**	**2209**	**0.99**	**0.00**	**2209**	**0.95**	**0.00**	**1039**

TABLE VII
PEARSON CORRELATION BETWEEN THE NUMBER OF OBFUSCATED VARIABLES AND (MILLIONS OF) ARITHMETIC OPERATIONS.

that are logged when obfuscated code is executed with and increasing number of obfuscated variables. Table VII reports the analysis of correlation between the number of encoding and decoding logged during execution and the number of obfuscated variables (IENC & IDEC versus NOBV), where IENC & IDEC are expressed in millions of operations.

Looking at Table VII, we can notice that the number of encodings/decodings is similar between code obfuscated with XOR Masking and with Merge Variables. In fact these two obfuscation transformations both require to encode all the assignments to and decode all the uses of an obfuscated variable. The number of encodings/decodings in the code obfuscated Residue Number Coding is lower, because many operations can be performed in the encoded domain. Only a subset of them need encoding, i.e. when used with operations not supported by the homomorphic scheme. This experimental result confirms the rationale of using Residue Number Coding to achieve higher level of obscurity in the code, because obfuscated values need to be decoded back in the clear less times when using the homomorphic scheme. However, the ratio of supported operations depends on the particular program that is subject to obfuscation.

VI. DISCUSSION

A. Practical Implications

Here we formulate some considerations and practical implications that should guide a practitioner in the choice of the most appropriate data obfuscation.

Types of sensitive variables: An obvious consideration is the type of the program variable that need to be obfuscated. In case the variables to hide are integer, a developer should consider to use *XOR masking*, *Residue Number Coding*, *Split Variables* or *Merge Variables*. In case of strings or array the option is for *Convert Static Data to Procedure*, *Restructure array* (all the four variants) or *Reorder Array*. However, customizations and combinations of these protections are also possible, for example *XOR masking* can be applied element-wise to obfuscate the content of a string or of an array.

Hide data access or the data value: Different transformations adopt different strategies to obfuscate data, and a developer should select the approach that best fits their requirements. The obfuscation classified as *storage and encoding* (see classification in Figure 1) are meant to change how values are represented in memory. This strategy should be used when the value itself is the asset to protect, such as a cryptographic key a sensitive credential or the results of a critical computation (e.g., the outcome of the license verification).

Obfuscation classified as *Aggregation* and *Ordering* do not change the value representation, but the way values are accessed, for example by scrambling the order of the elements in an array. These approaches are meant to protect the algorithm used to access values, and the read/write protocol. So, these obfuscations should be adopted when data access strategy is very important, more than data values.

Performance constraints: The application domain could pose constraints on the maximum performance overhead allowed to the obfuscation. Embedded systems and smart phones are examples of memory limited devices, that often offer limited computation resources or strict timing constraints. In these contexts, computationally cheap obfuscations (such as *XOR masking* and *Merge Scalar Variables*) are preferable than those more computationally expensive (such as *Residue Number Coding*). The detailed measurement of memory and performance overhead in the experimental comparison section should be used as a guide for choosing which data obfuscation to deploy.

Security Requirements: The strategy to adopt should fit the level of security required by the application to protect. *Residue Number Coding*, *Split Variables* and *Merge Scalar Variables* support homomorphic operations, i.e. they are able to perform selected operations with obfuscated data, without the need to de-obfuscate them. In this way, the clear values are visible only at the end of the computation, e.g. when they are printed. However, a developer should check that the security critical section of program to protect actually operates only obfuscated values with the supported operations. Additionally, a developer should evaluate if the computation overhead caused by the higher obscurity of these approaches is compatible with the application execution constraints.

Manual effort: Some obfuscations are not fully automatic, because they require some preliminary work to prepare the code for the transformation. Refining the results of pointer analysis is a common step that many obfuscations require. Apart form this, more preparation is needed by *Residue Number Coding*, *Split Variables* and *Merge Scalar Variables* to check if the value ranges or value signs are compatible with the transformations. *Restructure Arrays*, instead, requires that array elements are accessed only by index. String specific obfuscations (*Convert Static Data to Procedure* and *Reorder Array*) require to turn strings into static constants. The former requires also a post-process manual step (or automatic patch) to free the memory allocated to the string.

B. Threats to Validity

For an experiment not involving human participants, there are two types of threats to validity to consider, and they are the internal (whether confounding factors can influence the findings) and the external validity aspects (whether results can be generalized).

Regarding the internal validity, we have had to make certain that when a relationship is observed between two variables, it is due to a causal relationship, and not caused by an external factor that is not controlled, controllable or measured. To achieve this objective, we have considered metrics of execution overhead (memory and time). Moreover, the conclusions have been drawn based on an objective statistical test.

Our implementation of obfuscation transformations could contain errors. However, when implementing them, we strictly followed the published algorithm description by the original authors. Moreover, we tested the obfuscated programs in order to verify that they were still running correctly.

Regarding the external validity, we have had to consider whether the observed causal relationships can be generalized outside the scope of the experiment. We have designed this experiment as objectively and generally as possible, involving different obfuscation algorithms and source code from different programs. Nonetheless, only further experiment with other obfuscator tools and more subject applications can confirm our findings.

VII. Related Work

Catalogues of obfuscation transformations (with an informal comparison) have been already presented [5], [18], [19], [20], [21]

Collberg et al. [5] extensively review several obfuscation techniques, their strengths and weakness against various attacking techniques, and provide a taxonomy both for data and for control-flow obfuscations.

Balakrishnan and Schulze [18] presented a survey of the literature about obfuscation techniques. In particular, they illustrate the most prominent works in the field from the following perspectives: (a) general obfuscation methods, (b) obfuscation against static analysis, (c) obfuscation against dissembling, and (d) differences in obfuscating benign and malicious code. You and Yim [19] wrote a brief survey of malware obfuscation techniques.

Majumdar et al. [20] present a brief survey of program obfuscation techniques particularly focussing on two control-flow obfuscation techniques. Eventually Lai et al. [21] compared of 13 obfuscation tools for Java.

An empirical point of view is assumed in other works, mainly using source code metrics. However, their objective was mainly to measure the effect of *code* obfuscation on program statements, rather than the effect of *data* obfuscation on data structures.

The most related work on the assessment of code obfuscation has been presented by Heffner et al. [22]. They used metrics for obfuscation potency and performance degradation as they aimed at finding the optimal sequence of obfuscations to be applied to different parts of the code in order to maximize complexity and reduce performance overhead. With a similar goal, Jakubowski et al. [23] presented a framework for iteratively combining and applying protection primitives to code. They also considered code size, cyclomatic number and knot count metrics to evaluate the code complexity.

Collberg *et al.* [5] proposed the use of complexity measures in obfuscation tools to help developers choose among different obfuscation transformations. A high-level approach has been proposed by Collberg et al. [24] when they defined the concepts of *potency* and *resilience* of obfuscation. Karnick et al. [25] defined more precise metrics for potency (combining nesting, control-flow and variable complexities), resilience (as the number of errors generated decompiling obfuscated code) and cost (as an increment of memory usage).

Many authors have chosen just a few particular metrics with the assumption that these were good indicators of software complexity and ensure a harder task for the attacker when they try to break the code. Anckaert *et al.* [26] attempted to quantify and compare the level of protection of different obfuscation techniques. In particular, they proposed a series of metrics based on *code*, *control flow*, *data* and *data flow*: they computed such metrics on some case study applications (both on clear and obfuscated code), however without performing any validation on the proposed metrics. Linn et al. [27] define the *confusion factor* as the percentage of assembly instructions in the binary code that cannot be correctly disassembled by the disassembler, assuming that the difficulty of static code analysis will increase with this metrics, even if it strongly depends on the disassembly tools and algorithms used.

Later, a broad study [28] have been conducted to compare the effect of 44 obfuscations on 4 millions line of code with 10 metrics, including modularity, complexity and size metrics, using statistically sound evaluation.

VIII. Conclusion

This paper presents a survey data obfuscation transformations, listing the features of each obfuscation approach, applicability conditions, memory and execution time overhead.

Based on the analytical and experimental comparison, we distilled considerations and guidelines for the practitioners who are facing the challenge of deciding whether to adopt data obfuscation and, in case, which transformation to deploy.

Acknowledge

The research leading to these results has received funding from European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 609734.

References

[1] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski, "Guest editors' introduction: Software protection," *Software, IEEE*, vol. 28, no. 2, pp. 24–27, 2011.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," in *Advances in cryptologyCRYPTO 2001*. Springer, 2001, pp. 1–18.

[3] G. Wroblewski, "General method of program code obfuscation (draft)," Ph.D. dissertation, Citeseer, 2002.

[4] E. Lafortune *et al.*, "Proguard," *h ttp://proguard. sourceforge. net*, 2004.

[5] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Dept. of Computer Science, The Univ. of Auckland, Technical Report 148, 1997.

[6] C. Zarate, S. L. Garfinkel, A. Heffernan, K. Gorak, and S. Horras, "A survey of xor as a digital obfuscation technique in a corpus of real data," DTIC Document, Tech. Rep., 2014.

[7] J. Cannell, "Obfuscation: Malwares best friend," March 2013. [Online]. Available: http://blog.malwarebytes.org/intelligence/2013/03/obfuscation-malwares-best-friend/

[8] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[9] C. Collberg, S. Martin, J. Myers, and J. Nagra, "Distributed application tamper detection via continuous software updates," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 319–328. [Online]. Available: http://doi.acm.org/10.1145/2420950.2420997

[10] W. Zhu and C. Thomborson, "A provable scheme for homomorphic obfuscation in software security," in *The IASTED International Conference on Communication, Network and Information Security, CNIS*, vol. 5, 2005, pp. 208–212.

[11] H. L. Garner, "The residue number system," *Electronic Computers, IRE Transactions on*, vol. EC-8, no. 2, pp. 140–147, June 1959.

[12] C. Collberg, C. Thomborson, and D. Low, "Breaking abstractions and unstructuring data structures," in *Computer Languages, 1998. Proceedings. 1998 International Conference on*. IEEE, 1998, pp. 28–38.

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

[14] W. Zhu, C. D. Thomborson, and F.-Y. Wang, "Obfuscate arrays by homomorphic functions." in *GrC*, 2006, pp. 770–773.

[15] B. F. Demissie, "Implementation and assessment of data obfuscation for c/c++ code based on residue number coding," Master's thesis, University of Trento, 2014.

[16] P. Anderson and T. Teitelbaum, "Software inspection using codesurfer," in *Proceeding of the First Workshop on Inspection in Software Engineering*, Paris, France, july 2001.

[17] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow, "Txl: A rapid prototyping system for programming language dialects," *Computer Languages*, vol. 16, no. 1, pp. 97–107, 1991.

[18] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey," *CS701 Construction of Compilers*, vol. 19, 2005.

[19] I. You and K. Yim, "Malware obfuscation techniques: A brief survey." in *BWCCA*, 2010, pp. 297–300.

[20] A. Majumdar, C. Thomborson, and S. Drape, "A survey of control-flow obfuscations," in *Information Systems Security*. Springer, 2006, pp. 353–356.

[21] H. Lai, "A comparative survey of java obfuscators available on the internet," *Project Report, University of Auckland*, 2001.

[22] K. Heffner and C. Collberg, "The obfuscation executive," in *Information Security*. Springer, 2004, pp. 428–440.

[23] M. H. Jakubowski, C. W. Saw, and R. Venkatesan, "Iterated transformations and quantitative metrics for software protection." in *SECRYPT*, 2009, pp. 359–368.

[24] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation: tools for software protection," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 735–746, August 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=636196.636198

[25] M. Karnick, J. MacBride, S. McGinnis, Y. Tang, and R. Ramachandran, "A qualitative analysis of java obfuscation," in *Proceedings of 10th IASTED International Conference on Software Engineering and Applications, Dallas TX, USA*, 2006.

[26] B. Anckaert, M. Madou, B. D. Sutter, B. D. Bus, K. D. Bosschere, and B. Preneel, "Program obfuscation: a quantitative approach," in *QoP '07: Proc. of the 2007 ACM Workshop on Quality of protection*. New York, NY, USA: ACM, 2007, pp. 15–20.

[27] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*, ser. CCS '03. New York, NY, USA: ACM, 2003, pp. 290–299. [Online]. Available: http://doi.acm.org/10.1145/948109.948149

[28] M. Ceccato, A. Capiluppi, P. Falcarin, and C. Boldyreff, "A large study on the effect of code obfuscation on the quality of java code," *Empirical Software Engineering*, p. (to appear), 2014.