Assessment of Data Obfuscation with Residue Number Coding

Biniam Fisseha Demissie, Mariano Ceccato, Roberto Tiella

Fondazione Bruno Kessler, Trento, Italy
Email: {demissie,ceccato,tiella}@fbk.eu

Abstract

Software obfuscation was proposed as a technique to mitigate the problem of malicious code tampering, by making code more difficult to understand and consequently more difficult to alter.

In particular, "residue number coding" encodes program variables to hide their actual values, while supporting operations in the encoded domain. Some computations on encoded variables can proceed without the need to decode them back in the clear. Despite the obvious benefits of this approach, to the best of our knowledge, no implementation is available.

In this paper, we describe our implementation of data obfuscation based on residue number coding. Moreover, we present an assessment of this obfuscation scheme in terms of performance overhead, when more and more program variables are subject to obfuscation.

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. Code obfuscation prevents tampering by exploiting the economics of the attack. The objective of code obfuscation, in fact, is to obstruct code comprehension and turn attacks much more demanding and expensive for an attacker. In particular, code obfuscation is 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).

Residue Number Coding (RNC) is a form of data obfuscation that offers interesting mathematical properties that make it a good candidate for large adoption. Variables are encoded to hide original values, and they do not need to be decoded back in the clear when used in certain expressions, i.e. this obfuscation supports operations among obfuscated values. Though a similar obfuscation technque has been proposed based on a different encoding scheme [1], to the best of our knowledge, no implementation is available for RNC based obfuscation scheme and no empirical assessment has been conducted to study the impact of this obfuscation on real programs. Only theoretical evaluations have been proposed [2], [3], [4].

In this paper, we present our experimental quantification of data obfuscation based on residue number coding. First of all we present our implementation of this obfuscation scheme. Then, we propose our analysis of obfuscated code, based on the quantification of performance and memory overhead caused by obfuscation of different sets of variables. In future work, we intend to involve human participants to quantify the resilience of this obfuscation scheme against attacks, as we already did on other state-of-the-art obfuscations [5].

II. Data Obfuscation

Code obfuscation is the process of transforming a program P into another semantically-equivalent program P' so that P' is harder to understand than P according to a given definition of "understandability".

Among the possible transformations that can be applied to (the source code of) a program to obfuscate its semantics, *data obfuscation* transformations aim at hiding data values by changing the statements where variables are defined and used. Known data obfuscation transformations comprise techniques such as change variables scope from local to global, split or merge variables (both scalar and arrays), move constants from data to behavior, and change encoding [6].

In detail, *Change Encoding* Transformations (CET) modify how values are represented. Some transformations might involve only value representation without changing how values are stored in memory. As a trivial example, the affine CET that changes the representation of integers such that a value k is encoded as $k' = 2k + 1$, does

not require a type change provided that k' fits the size of an integer. Other CETs might actually modify how the memory is allocated, for example "Merge variables" is a transformation that pairs two integer variables into a single long variable.

A CET is a syntactical transformation that, applied to a program, modifies variable declarations, initializations, definitions (i.e., assignments statements), variable uses and the representation of constants. However to fully grasp the implication of applying a CET, it is useful to study the bijective mathematical function $e : T \to T'$ induced by the transformation from the set T of original values (also called the *clear* domain) to the set T' of encoded values (also called the *encoded* domain). Without any risk of ambiguity in what follows, $e(\cdot)$ will also denote the syntactical transformation on the source code. The inverse function $e^{-1}(\cdot)$ of $e(\cdot)$ is commonly denoted by $d(\cdot)$.

In the rest of this section, to ease the reader in following the presentation of our work, we will adopt the example presented in Figure 1(a). This is a fragment of C code that contains different variables. Let's assume that for a subset of them, namely *a*, *b* and *c*, tampering does not represent a threat. While, the other variables, namely *w*, *x*, *y* and *z*, are security sensitive and they need to be protected with obfuscation.

A. Base Transformation

If no further assumptions are made, a transformation $e(\cdot)$ is applied to source code as follows (*base transformation*):

- When a result of the evaluation of an expression `<exp>` is assigned to an encoded variable `<var>`, `<exp>` must be encoded, i.e., a statement like

  `<var>=<exp>;`

  becomes

  `<var>=e(<exp>);`
- When an encoded variable `<var>` is used, the decode function d has to be applied to its value, so that every use of `<var>` is replaced by `d(<var>)`;

Thus, for example, applying a base transformation to a statement like:

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

where x, y and z are encoded, the transformed version will have the form:

```
z = e(d(x)+d(y));
```

and, detailing the example, if $e(\cdot)$ is the affine transformation mentioned before, namely $e(\nu) = 2\nu+1$, the same line will be:

```
z=2*((x-1)/2+(y-1)/2)+1=x+y-1;
```

B. Homomorphic Transformations

A function $e : T \to T'$, where T is equipped with an operation $\oplus$ and T' with a corresponding operation $\oplus'$, is called *homomorphic* [7], if for all $\nu_1, \nu_2 \in T$, the following condition holds:

$$e(\nu_1 \oplus \nu_2) = e(\nu_1) \oplus' e(\nu_2). \quad (1)$$

For example, the linear transformation $l : \texttt{int} \to \texttt{int}$, defined by $l(\nu) = 2\nu$ can be easily proved homomorphic with respect to addition:

$$l(\nu_1 + \nu_2) = 2(\nu_1 + \nu_2) = 2(\nu_1) + 2(\nu_2) = l(\nu_1) + l(\nu_2).$$

If the encoding function $e(\cdot)$ is homomorphic then the result of the base transformation can be simplified, leveraging implications of Equation 1, for example:

$$z = e(d(x) \oplus d(y)) = e(d(x)) \oplus' e(d(y)) = x \oplus' y.$$

Thus if a transformation is homomorphic with respect to certain operations, transformed programs will present these operations performed directly in the encoded domain, avoiding to expose clear values.

C. Residue Number Coding

Residue Number Coding is a CET based on the Residue Number System. Residue Number System was introduced by Garner to improve performance of circuitry for multiplication of integers [8]. RNC takes an integer value ν and splits it in $u > 0$ components:

$e(\nu) = (\mu_1, \mu_2, ..., \mu_u)$

where $\mu_k = \mod(\nu, m_k)$, with $m_1, m_2, ..., m_u$ pairwise co-prime[1] positive integers. Provided that $0 \leq \nu \leq M$ where $M = m_1 \cdot m_2 \cdot ... \cdot m_u$ and under the hypothesis that $m_1, m_2, ..., m_u$ are pairwise co-prime, the "Chinese Remainder Theorem" ensures that $e(\cdot)$ is bijective. The decoding function $d(\mu_1, \mu_2, ..., \mu_u)$ can be computed using the Euclid's extended greatest common divisor algorithm [8].

Operations in the encoded domain are defined "per component", i.e.:

$(\mu_1, \mu_2, ..., \mu_u) +' (\lambda_1, ..., \lambda_u) = (\mu_1 + \lambda_1, ..., \mu_u + \lambda_u)$

$(\mu_1, \mu_2, ..., \mu_u) \cdot' (\lambda_1, ..., \lambda_u) = (\mu_1 \cdot \lambda_1, ..., \mu_u \cdot \lambda_u)$

It can be shown that the transformation $e(\cdot)$ is homomorphic both with respect to addition (consequently subtraction) and multiplication. It is worth noting that, as

$\mod(\mu_k + r_k m_k, m_k) = \mod(\mu_k, m_k)$

for any integer r_k, to improve the strength of the transformation, a known strategy is to add to each component μ_k a term $r_k m_k$ where r_k is randomly chosen. For

[1]Two integers m and n are co-prime if $gcd(m, n) = 1$

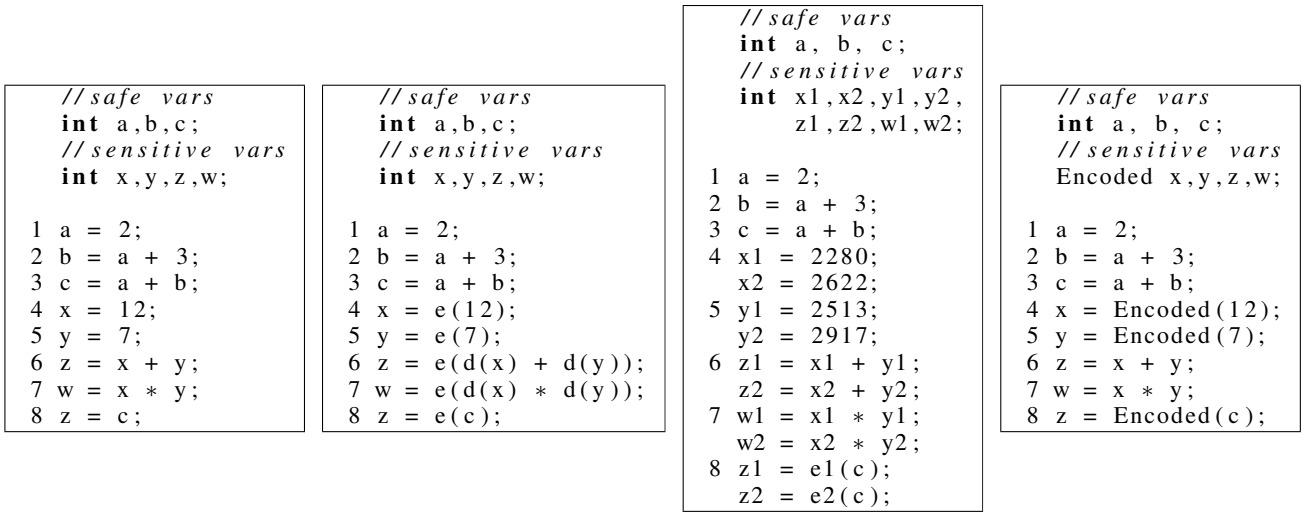

Fig. 1. Running example of C code: (a) original source code, (b) intermediate transformation step, (c) obfuscated version, and (d) final result with support class and operator overloading.

example, if $u = 2$ and $m_1 = 14$ and $m_2 = 15$, then every $x \in 0, 1, ..., 209$ can be represented as:

$e(\nu) = (\text{mod}(\nu, 14) + 14r_1, \text{mod}(\nu, 15) + 15r_2)$

Therefore, the program in Figure 1(a) is transformed by means of $e(\cdot)$ to the program in Figure 1(c). For the sake of clarity, an intermediate state of the transformation is shown in Figure 1(b). In the first step (Figure 1(a) to Figure 1(b)), encoding and decoding functions are applied to definitions and uses of all sensitive variables, namely w, x, y and z, and constants 12 and 7. In the second step, every statement that contains an assignment to an encoded variable is split in two to manage the two components (line 4, 5, 6, and 7). Constants '12' in line 4 and '7' in line 5 are encoded as (2280,2622) and (2513,2917) respectively. Statements involving addition and multiplication of encoded values can be simplified due to the homomorphic property of RNC (line 6 and 7) while assignment to variable 'z' at line 8 requires two function calls to encode the decoded (clear) value of variable 'c'.

D. Transformation

We implemented data obfuscation based on RNC (of two components) as a source code transformation in Clang-LLVM [9]. LLVM was developed as compiler with the underlying idea of implementing all the optimizations on an intermediate language (the LLVM bitcode). A frond-end translates source code to bitcode and a back-end translates the (optimized) bitcode to binary. Clang is the LLVM front-end for C/C++. We relied on the parse tree available in Clang to implement data obfuscation transformation rules.

The obfuscation transformation requires as input the names of variables to be encoded with RNC. They are identified using a configuration file, that lists the (fully referenced) list of variables to obfuscate, one per line. Function local variables are prefixed with the name of function where they are defined, while global variables are prefixed with the keyword `global`.

Instead of doubling all those expressions that involve obfuscated data, as presented in the mathematical background (see Figure 1(c)), we adopted a support class and we relied on operator overloading. This class is called *Encoded* and two class fields (x and y) store the values of the two components. Constructors and utility functions are provided to encode/decode values. Operators are overloaded to support operations in the encoded domain as operations among instances of the *Encoded* class.

First of all, in the declaration of a sensitive variable, the type is changed from *int* to *Encoded*, see declarations in Figure 1(d).

Then, assignment expressions are changed according to the type of the variable in the left-hand side (clear or encoded). If the type of left-hand side variable is in the clear domain (e.g., `int`) and the type of the right-hand side is encoded, then the expression in the right-hand side is decoded before the assignment using the $d(\cdot)$ operator (introduced in Section II-C). No change is required for statements 1, 2 and 3 because the right-hand side expressions are already in the clear domain.

Conversely, if the variable in left-hand side of the assignment is of type *Encoded*, then all clear variables and expressions in the right-hand side of the assignment

are changed to the encoded domain using the $e(\cdot)$ operator. This is the case of statements 4, 5, 6, 7 and 8.

Operations supported by the RNC scheme do not need to be updated thanks to the operator overloading. In fact, the sum among integers and the sum among *Encoded* variables adopt the same syntactic notation (i.e., +). This is the case of statements 6 and 7, where the sum symbol is mapped to the operator defined in the class *Encoded*. Conversely, statements 4, 5 and 8 require explicit conversion.

RNC is homomorphic w.r.t. multiplication, sum and subtraction, and all the operations that can be expressed as a composition of them (e.g., left shift of n positions is equivalent to a multiplication by 2^n). All the remaining operations must be performed in the clear, so encoded value should be decoded, operated and then encoded back.

The final result of the transformation is shown in Figure 1(d). Even if the transformation is described and conducted in terms of source code, the compiled binary version will be delivered and will be potentially subject to attacks.

III. Empirical Validation

The most conservative strategy to 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 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 variable(s) that is(are) security sensitive and is(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.

The data-obfuscation tool is executed on several obfuscation configurations, resulting in many different obfuscated versions of the same program[2].

We expect memory overhead due to the fact that each obfuscated variable is split and represented as two variables. Computational overhead is expected, because operations among encoded variables should be performed twice, once per each of the two dimensions. Moreover, additional computational overhead is required when obfuscated variables have to be encoded/decoded each time an operation is performed that is not supported by the encoding scheme (e.g., division).

Thus, we formulate the subsequent research question for our empirical assessment:

- **RQ**$_1$ What is the *memory* overhead due to data obfuscation with residue number coding when increasing the number of obfuscated variables;
- **RQ**$_2$ What is the *execution time* overhead due to data obfuscation with residue number coding when increasing the number of obfuscated variables.

A. Metrics

The definition of metrics to consider derives directly from the research questions. They are:

- **Memory**: The total amount of memory used by the process to execute a scenario;
- **Time**: The total amount of time taken by the process to execute a scenario;
- **Number of Obfuscated Variables**: The number of variables that are subject to data obfuscation.

The execution environment is instrumented to measure the first two metrics. In particular *Memory* is measured using the unix *Time* utility. We used the command `Time -f "%M"` to quantify the maximum resident set size of the process during its lifetime. This utility also supports the measurement of the *Time* taken by a process, however at a too coarse grain resolution (seconds). A more fine grain measurement can be achieved by marking the system time when the process starts and ends. Using system time, we could reliably[3] measure the execution time with the granularity of the millisecond.

Obscurity of a program code is measured by the number of program variables that are subject to data obfuscation.

B. Case studies

To answer research questions RQ_1 and RQ_2 we consider two case study applications, *license check* and *MD5*. The first case study, *license check*, is a small routine devoted to check the validity of a license number to activate a software component. The serial number contains the date when the license has been emitted and its validity is meant to expire 30 days after emission.

The security sensitive variable is the one that holds the difference 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

[2]The complete experimental setting (tools and case studies) is available for replication purposes at http://selab.fbk.eu/spro2015/

[3]To make the measurement reliable, no other process was executed during the experiment and the machine was disconnected from the network.

value to 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, e.g., 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 protection are effective against these attacks, such as code obfuscation or remote attestation.

The difference between current date and date from license should be obfuscated. However, other variables involved in the computation might leak sensitive information, such as dates, days, months, years, and they are also candidate for obfuscation according to their distance to the sensitive variable.

The second case study is *MD5*, a routine to compute the checksum of files. This program has been selected because we meant to study the effect of data obfuscation on an I/O intensive example. To this aim we identified the sensitive variable as the one that stores and accumulates the checksum value.

C. Experimental Setting

To study the impact of obfuscation, a set of execution scenarios have been defined for the two case studies. For license check, we defined 365 scenarios, corresponding to valid licenses emitted on consecutive dates. Part of the licenses are valid, the rest of them are expired.

The execution of this program is quite fast, and the time to initialize the process could dominate the time to execute the process. To avoid this problem, we artificially modified the program by introducing an iteration of 1000 executions of the *main* function. In this way, the program execution time dominates the total time and any variation of execution time due to obfuscation would be more evident. In this context license check can be considered a cpu-intensive case study.

For MD5, we defined 4 scenarios, that consist in 4 files of increasing size for which to compute the checksum. The files are installation packages for Spin (2Mb), Skype (17 Mb), Chrome (36 Mb) and Eclipse (145 Mb). To avoid the bias effect of non-deterministic event during the experiment, each execution is repeated 10 times.

The experimental process consists of running the original (clear) code on the execution scenarios, to collect *Memory* and *Time*. Then, for increasing number of variables to be obfuscated we apply data obfuscation, we execute the obfuscated code and we collect *Memory* and *Time* 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 using the unix *diff*. In this way we test the obfuscated code for correct behaviour, because we need to be confident that the transformations applied by our tool did not introduce errors or deviations on the original functionalities.

The experiment has been conducted on a laptop with Intel Core i3 2.3 GHz CPU (4 cores), 4 GB of memory, running Ubuntu 12.10 64 bit.

D. Results

The trend of memory consumption on *license check* is shown in Figure 2. The first box in the plot represents the clear code, where no variable is obfuscated. More and more variables are obfuscated, respectively 2, 7 and 8 variables. Consequently, more memory is required to run the program.

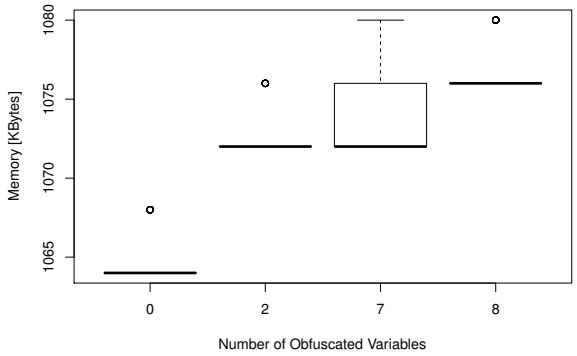

Fig. 2. Boxplot of memory consumption (license check).

Figure 3 shows the boxplot of the time required to run the program. Also in this case, the more variables are obfuscated, the more time is taken by the execution.

To test if the trend observed on the graphs is statistically significant, we use the *Pearson correlation* test to check if *Memory* and *Time* are correlated with the *Number of Obfuscated Variables*. This test computes the correlation coefficient r, a symmetric, scale-invariant measure of association between two random variables. It ranges from -1 to +1, where the extremes indicate perfect (positive or negative) correlation and 0 means no correlation. It also computes a *p-value*, and we observe statistical significance with a confidence of 95% when p-value is <0.05. In case of statistical significant correlation, we provide also the linear coefficient of the linear interpolation.

Table I reports the results of the statistical test for *license check*, with statistical significant cases highlighted

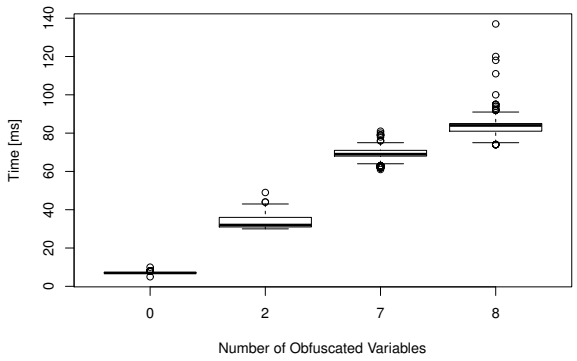

Fig. 3. Boxplot of execution time (license check).

in boldface. As observed on the graphs, both *Memory* and *Time* report a significant correlation with the *Number of Obfuscated Variable* (p-values $<$0.05, and r is near to 1). The linear coefficients allow to quantify the obfuscation overhead for the first case study. When adding a new variable to the list of variables to obfuscate, we expect that the process requires additional 1.19 KBytes of memory and 8.85 milliseconds of time to complete the execution.

TABLE I. Correlation between obfuscation and cost (license check).

Metrics	P-value	Correlation	Coefficient
Memory & # of Obf. Variables	$<$**0.01**	0.85	1.19
Time & # of Obf. Variables	$<$**0.01**	0.98	8.85

To understand if these observations hold also for an I/O intensive program, we repeat the same experiment with the second case study, i.e., *MD5*. We run this program with 4 distinct input of increasing size.

The results of the Pearson correlation for *MD5* are reported on Table II. Different execution scenarios are analyzed separately. In the analysis of *Memory*, shown in Table II(a), we observe statistical significance in three out of four cases, however the r is fairly far from perfect correlation (case of r=1). In the significant cases, the memory increase per obfuscated variable is approximately of 1 KByte.

Differently from *license check*, for *MD5* the memory consumption is quite flat, with no apparent effect due to obfuscation. The only noticeable difference between executions seems to be due to the size of the input data, i.e., most of the memory is allocated by the I/O. The incremental fraction of memory required by the obfuscation is negligible with respect to existing program memory requirements.

The analysis of the correlation between obfuscation and execution time is shown in Table II(b). None of the execution scenario reached statistical significance (p-value is never $<$0.05), execution time and number of obfuscated variables look non-correlated (r near to 0).

IV. Related Work

Data obfuscation transformations are described and qualitatively analyzed in various previous works (e.g. [10] and [6]). Drape et al. [2] describe how data obfuscation transformations can be formalized using data refinement: a setting that allows to formally prove transformations correctness. Residue Number Coding and its homomorphic properties are presented in a work by Zhu, W. and Thomborson C. [3]. Their study concerns theoretical properties of the RNC transformation. The same authors published an article on applying homomorphic data obfuscation to array indexing [4]. All these works present definitions, applications and qualitative analyses for data obfuscation techniques in general, and homomorphic data obfuscation in particular, but they lack to address implementations and empirical assessment to evaluate effective applicability of data obfuscation in the real world.

In the past, the assessment of obfuscation transformations has been conducted by measuring the complexity introduced by obfuscation mainly through 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 and Collberg [11]. 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. [12] 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. [6] 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. [10] when they defined the concept of *potency* of an obfuscation as the ratio between the complexity (measured with any metric) of the obfuscated code and the complexity of the original source code, and the concept of *resilience*, i.e., how difficult is to automatically de-obfuscate the protected code. Karnick et al. [13] defined more precise metrics

TABLE II. Correlation between obfuscation and cost (MD5).

Input	P-value	Correlation	Coefficient
2 Mb	<**0.01**	0.58	1.16
17 Mb	0.07	0.29	-
36 Mb	<**0.01**	0.66	1.05
145 Mb	**0.03**	0.34	0.72

(a) Memory & Number of Obfuscated Variables

Input	P-value	Correlation	Coefficient
2 Mb	0.61	-0.08	-
17 Mb	0.84	0.03	-
36 Mb	0.06	-0.30	-
145 Mb	0.35	0.15	-

(b) Time & Number of Obfuscated Variables

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. [14] 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. [15] 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 [16] have been conducted to compare the effect of 44 obfuscations on 4 millions line of code with 10 metrics, including (Chidamber and Kemerer) modularity, (cyclomatic) complexity and size (lines of code). In order to provide reliable results, a statistically sound evaluation has been conducted.

V. Conclusion

This paper presents our implementation of data obfuscation with residue number coding. It also presents a study to analyze how performance degrades when obscurity increases (i.e., the number of obfuscated variables). Results suggest that this obfuscation scheme is quite lightweight, because it does not impose dramatic costs in terms of computation and memory overhead.

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] "The tigress diversifying c virtualizer," http://tigress.cs.arizona.edu/#data, 2015.

[2] S. Drape, C. Thomborson, and A. Majumdar, "Specifying imperative data obfuscations," in *Information Security*. Springer, 2007, pp. 299–314.

[3] 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.

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

[5] M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques," *Empirical Software Engineering*, vol. 19, pp. 1040–1074, 2014.

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

[7] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.

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

[9] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[10] 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

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

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

[13] 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.

[14] 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.

[15] 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

[16] 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.