

Assessment of Source Code Obfuscation Techniques

Alessio Viticchié*, Leonardo Regano*, Marco Torchiano*, Cataldo Basile*,
Mariano Ceccato†, Paolo Tonella† and Roberto Tiella†

*Dipartimento di Automatica e Informatica,
Politecnico di Torino, Torino, Italy
{first.last}@polito.it

†FBK, Trento, Italy
{last}@fbk.eu

Abstract—Obfuscation techniques are a general category of software protections widely adopted to prevent malicious tampering of the code by making applications more difficult to understand and thus harder to modify. Obfuscation techniques are divided in code and data obfuscation, depending on the protected asset. While preliminary empirical studies have been conducted to determine the impact of code obfuscation, our work aims at assessing the effectiveness and efficiency in preventing attacks of a specific data obfuscation technique – VarMerge. We conducted an experiment with student participants performing two attack tasks on clear and obfuscated versions of two applications written in C. The experiment showed a significant effect of data obfuscation on both the time required to complete and the successful attack efficiency. An application with VarMerge reduces by six times the number of successful attacks per unit of time. This outcome provides a practical clue that can be used when applying software protections based on data obfuscation.

I. INTRODUCTION

Recently, a new class of attacks against software has received great attention: Man-At-The-End attacks (MATE) [1], [9]. MATE attacks aim at compromising software assets, which can be classified in two types: data and code. Software developers aim at protecting assets' security properties, namely data confidentiality, privacy, integrity, and code confidentiality, execution correctness, and integrity. MATE attacks are more powerful than the ones against traditional cryptography, the Man-In-The-Middle attacks, as MATE attackers are the end users. Hence, they have full privileges and full access to executables on their platforms, where they also have at their disposal a vast set of tools, like debuggers, decompilers, and static and dynamic analysers.

Obfuscation is one of the most used protection technique to prevent the comprehension of programs against MATE attacks. There are several approaches to obfuscation [7], which can be mainly divided in code obfuscation and data obfuscation. All obfuscation types transform a program in such a way that it is more difficult to understand for an attacker, while at the same time preserving its original program functionality. However, data obfuscation transformations change programs with the aim of hiding both variable content and usage, while code obfuscation changes layout and control information to render

the code difficult to reverse engineer and understand. The hypothesis is that obfuscation can protect asset confidentiality by making it difficult for attackers to understand the application under attack. Obfuscation is also used indirectly to protect data and code integrity, as well as execution correctness, since changing programs' functionality is also difficult when the code is difficult to understand. Ultimately, obfuscation discourages attacks against software assets as it renders them less economically favourable. Indeed, mounting successful attacks against software assets protected with obfuscation requires more sophisticated attack tools, and more time to comprehend the assets, thus a greater investment. As a consequence, profits from exploiting software attacks are reduced.

Obfuscation makes attacks *more* complex but it cannot completely block them [3]. Researchers have focused their attention on how much obfuscation is effective in protecting software assets. Initially, the impact of obfuscation has been extensively investigated by means of traditional software assessment techniques, based on internal code metrics [8], [2], [12], [11], [16], [17], [4]. The claim is that by means of these methods an objective evaluation of tamper resistance can be reached. However, these types of quantification of software protections are unable to entirely capture the actual complexity for attackers who aim at compromising the assets. A program with higher code complexity (e.g., one protected with control flow obfuscation) should require at least the same time to mount attacks, but there is no evidence of significant correlation between complexity metrics and actual attack delays. In other words, impact of code protections metrics was not empirically validated.

Later, empirical studies have been conducted [15], [6], [5] to assess the impact of code obfuscation on delaying the completion of attack tasks. These studies have targeted a quantitative estimation of the cost of understanding obfuscated code, by means of controlled experiments with human subjects. Human subjects have been asked to perform tasks on ad hoc defined and protected applications. Then, the impact of code obfuscation has been estimated by means of two main parameters: *correctness*, which counts the number of successful attacks, and *efficiency*, which measures the delay

added by the presence of obfuscation.

This paper assesses the effectiveness and efficiency of the VarMerge data obfuscation technique by comparing the time needed to mount attack tasks on clear and obfuscated versions of two applications written in C, and assessing the success rate in the execution of the task. The VarMerge technique has been selected among a set of candidate techniques as one of the most effective ones (using Collberg’s terminology, with high potency) and is applicable to C source code (a prerequisite for the involvement of students as subjects at our institutions). While it is a general feeling that obfuscation renders programs more difficult to understand, we wanted to confirm this hypothesis on a data obfuscation technique and derive, if possible, quantitative measures of its impact. To the best of our knowledge, this is the first work that aims at estimating with an empirical study the effectiveness of a data obfuscation technique. While previous work focused on control-flow obfuscation.

The paper is organised as follows. Section II presents previous works on the assessment of obfuscation effectiveness. Section III details the experiments preparation, realisation, and analysis method. In Section IV, the results of the experiments are analysed to confirm or reject the research hypothesis. Section V discusses the results and the limitations that might affect their validity. Finally, Section VI draws conclusions and presents future experiments that may lead to a more comprehensive assessment of obfuscation techniques.

II. BACKGROUND

A. Related work

There are two main research approaches for the assessment of obfuscation techniques: assessment based on internal software metrics and assessment with experiments involving human subjects.

The first approach has been introduced by Collberg *et al.*, who defined the concept of *potency*, a metric to estimate the effectiveness of obfuscation on programs [8]. Potency has been later used by Anckaert *et al.* to compare obfuscation techniques [2]. Linn *et al.* considered the confusion factor, which estimates the number of binary instructions that a code decompiler is not able to parse [12]. Goto *et al.* proposed a method to quantitatively measure the complexity of obfuscated code based on the compiler syntax analysis [11]. Udupa *et al.* estimated the increase of complexity in obfuscated programs by using data that can be extracted with static and dynamic analysis tools [16]. Visaggio *et al.* instead used code entropy as a protection potency metric for obfuscated Javascript code [17]. Ceccato *et al.* evaluated the impact of several obfuscation techniques on Java code quality [4]. The authors performed a large set of experiments and estimated the effects of obfuscation on ten different complexity and modularity metrics.

Assessment by means of experiments with human subjects has been first presented in a work by Sutherland *et al.*, who published the first study with human subjects [15]. The authors correlated the expertise of attackers with the correctness of reverse engineering tasks. Moreover, they proved that source

code metrics are not appropriate to estimate the delays on attack tasks, when binary code is involved. Ceccato *et al.* measured, with two controlled experiments, the correctness and effectiveness in understanding and modifying decompiled obfuscated Java code, compared to decompiled clear code [6]. This work has been extended with a larger set of experiments on several obfuscation techniques in a successive work [5]. The major difference with respect to previous work is the obfuscation used in the study. In fact, Ceccato *et al.* studied obfuscations aiming at hiding control flow and variable names. Conversely, we focus on a transformation meant to hide values of critical variables. Our work continues the effort in empirically assessing the effectiveness of protection techniques by means of experiments involving human subjects, by addressing a technique, VarMerge, which was not assessed before, in a category of obfuscation techniques, data obfuscation, which was not yet target of experiments.

B. Obfuscation

As anticipated, data obfuscation aims at hiding both variable content and usage. Data obfuscation can be applied for instance to critical data, such as: user-IDs, counters, expiration dates, or privacy-sensitive data such as medical data. On the other hand, data obfuscation is not suited for hiding cryptographic keys, as in this case white box cryptography offers a stronger protection for this specific asset. Several data obfuscation transformations have been proposed in the literature. They have been initially classified as [8]:

- **Storage & Encoding:** change representation of (scalar) data;
- **Aggregation:** alter how data (both scalar variables and arrays) are aggregated;
- **Ordering:** permute items in existing data structures.

In this paper, we report the empirical validation of the effectiveness of a specific aggregation technique, namely VarMerge, which merges several scalar variables into a single one. When selecting this technique, we have only considered Storage & Encoding and Aggregation techniques that are not targeted for the Object Oriented paradigm. We have excluded Reordering transformations as their potency is low. The candidate techniques were: ‘Split variables’, ‘Change encoding’, ‘Change variable lifetimes’, ‘Convert static data to procedure’, VarMerge, and ‘Split, fold, merge, arrays’. All the advantages and disadvantages of these techniques have been considered and, in the end, VarMerge has been selected because of its high potency, the reasonable effort to implement an automatic data obfuscator, and because it can be applied to the source code of applications written in C, the language mastered by students at our institutions. All these decisions have been made and validated within the context of the ASPIRE project¹.

Given n variables v_1, \dots, v_n whose domain is represented by $b(v_1), \dots, b(v_n)$ bits, VarMerge creates a new variable m ,

¹www.aspire-fp7.eu

clear.c	obfuscated.c
<pre> 1 int main() { 2 int a, b; 3 4 a = 3; 5 b = 5; 6 7 printf("%d\n", a+b); 8 } </pre>	<pre> 1 int main() { 2 unsigned long x; 3 4 /* 3L (5L << 0x20) */ 5 x = 21474836483L; 6 7 printf("%d\n", (int) (x&0xffff)+(int) (x>>0x20)); 8 } </pre>

Fig. 1: Example of clear and obfuscated code using VarMerge

whose domain is $b(m) = \sum_i^n b(v_i)$, VarMerge performs the following aggregation:

$$m = v_1 + v_2 \cdot 2^{b(v_1)} + \dots + v_n \cdot 2^{\sum_{i=1}^{n-1} b(v_i)}$$

Operations on the original variables can be mapped to operations on the merging variable by means of proper mask and shift operations.

Figure 1 reports an example of clear code and the corresponding obfuscated code. We can observe how the original variables as well as their values is no more clearly available.

The effectiveness of VarMerge relies on the fact that it breaks the easy association between a variable and its use, that is, its semantics. Since most of the operations in the obfuscated code involve the use of the same variables with different shifts, an attacker is obliged to understand the code semantics by reconstructing the variable flows and by associating values with shifts. Moreover, proper bitwise operations are needed to obtain the actual decimal values. Of course, expert attackers may recognise that a program has been protected with VarMerge by the unusual number of shift operations. Correspondingly, they may adapt their attack strategy to the recognised protection. Nevertheless, they have to invest non-negligible time to analyse the aggregation variables and to determine the shift offsets necessary to reconstruct the program semantics.

III. EXPERIMENT DESIGN

The next sections present all the preparation and realisation phases of the experiment.

A. Goal and Research Questions

The main *goal* of the study is to evaluate the effect of a specific source code obfuscation technique, VarMerge, with the *purpose* of evaluating its ability of making the code resilient to malicious attacks. The *quality focus* is how the technique reduces the attacker’s capability to successfully perform an attack by forcing the application behaviour.

The study is interpreted from the *perspective* of an attacker, since we aim at evaluating the increased difficulty perceived by attackers when the code is protected by VarMerge. In particular, in our case the role of the attacker is played by a set of students that have a consolidated minimum level of expertise in manipulating application source code.

1) *The subjects*: The *context* of the study consists of *subjects*, i.e., the students acting as attackers, who perform their attacks on *objects*, i.e., the systems to be attacked.

Subjects are 15 University students: 14 Master students in Computer Science Engineering and 1 PhD student in Computer and Control Engineering, both from Politecnico di Torino. All the students are knowledgeable about C programming and software engineering. We filtered Master and PhD students based on their academic career and grades. Supported by the fact that Politecnico di Torino offers and requires a strong knowledge of programming, in particular C programming, in many courses, we estimated their expertise as sufficient to perform the tasks that we were proposing them. Moreover, we precisely estimated their expertise during the results assessment phase based on their expertise self estimation and number of years of experience, which is the best practice according to a previous work [10].

The subjects are not expected to have any knowledge about MATE scenarios, attacks, and attacks strategies. Indeed, the students of Politecnico di Torino do not attend any course about software tampering or software reverse engineering. Thus, students are probably not the best choice to model real subjects. Professional hackers could be better subjects to evaluate MATE attacks exploitation, but it is considerably difficult to involve them. We considered the competences and capabilities of our subjects during the design and the analysis, in particular when selecting the applications and the tasks. We think the use of students as subjects did not affect our main conclusions, as explained in Section V-B, since we measure the impact of VarMerge on attack time in a comparative way.

We encouraged the students’ participation by putting up for grab two gift certificates among all the participants regardless of the success in performing their attack tasks. In our opinion, the prize encourages participation and, at the same time, the possibility of a win induces a larger commitment in the tasks. On the other hand, assigning prizes to all participants would lead to higher participation, but it would introduce noise into the collected data, as subjects might participate just for prize. Finally, giving no incentives at all was not possible, as students do not usually like to spend extra time on non-profitable academic tasks instead of investing it in regular academic activities. In conclusion, university students participation had to be stimulated in some way, since they are students and not professional attackers (such as Tiger teams, penetration testers

or ethic hackers) as they do not get any monetary advantages in performing the experiment tasks.

2) *The objects*: The *objects* of the experiment are two applications written in C. We define as *clear* the original application with no obfuscation applied and as *obfuscated* the application on which VarMerge is applied.

The first application, named *Lotto*, is a stand-alone lottery game. This game allows the user to input a sequence of seven numbers (six numbers plus one bonus number) and tells, as output, if the sequence matches another sequence, named the jackpot sequence, which is hard-coded in an array variable inside the application source code. The original version of Lotto is a 238 LOC application. VarMerge obfuscation enlarges it to 291 LOC.

The second application, named *Lottery*, is a client-server lottery game similar to bingo that works as follows: in order to extract the bingo numbers, the client contacts the server asking for a challenge; the server generates a random sequence of bytes and sends it back to the client; the client uses the challenge as a random seed to derive a sequence of seven numbers whose value ranges between 1 and 39 - these are the seven extracted numbers. To claim a win, the customer exhibits its number sequence. The winning sequence is sent back to the server that checks its validity against the previously sent challenge; if the sequence is valid, the server accepts it and prints it in a log - so, the win can be delivered to the customer; otherwise, the server rejects the sequence and exits. This iteration is repeated ten times in the application to be attacked.

The original version of Lottery is composed of 62 client LOC and 498 server LOC. VarMerge obfuscation makes the client as large as 84 LOC and the server 521 LOC. The client uses a support library, for communication purposes only, consisting of 452 LOC. This library is not protected with VarMerge.

The complexity of the two systems has been designed to be different. In fact, despite the number of LOC, Lottery is harder to understand and to modify, because it involves server-side logic that cannot be inspected by the subjects.

Subjects are asked to carry out an *attack task*. For the Lotto application, the attack task is to determine the jackpot sequence, by spotting where the hardcoded array with the sequence is declared and defined, and reporting its values (it is unlikely that subjects are able to guess by random choice the winning sequence in the allotted time). For the Lottery application, the attack task is to modify the application to force the client to only extract numbers between 1 and 20. The attack tasks do not depend on clear and obfuscated application, they only depend on the application (Lotto or Lottery).

We overall identify the tasks as follows:

- $\mathcal{T}_{C,Lo}$ the attack ported on the *clear Lotto* application;
- $\mathcal{T}_{O,Lo}$ the attack ported on the *obfuscated Lotto* application;
- $\mathcal{T}_{C,Ly}$ the attack ported on the *clear Lottery* application;
- $\mathcal{T}_{O,Ly}$ the attack ported on the *obfuscated Lottery* application;

TABLE I: Task assignments

	First sub-session task	Second sub-session task
Group 1	$\mathcal{T}_{C,Lo}$	$\mathcal{T}_{O,Ly}$
Group 2	$\mathcal{T}_{O,Lo}$	$\mathcal{T}_{C,Ly}$
Group 3	$\mathcal{T}_{C,Ly}$	$\mathcal{T}_{O,Lo}$
Group 4	$\mathcal{T}_{O,Ly}$	$\mathcal{T}_{C,Lo}$

The experiment is performed in a unique session divided into two sub-sessions. Each subject undergoes two distinct tasks, one per sub-session. The session lasted 3 hours and 30 minutes, the sub-sessions have approximately lasted 1 hour and 45 minutes. The task assignments were fully balanced, to avoid the influence of obfuscation and application across sub-sessions: each subject never worked on neither the same application nor the same treatment (obfuscation) in the two subsequent tasks. Therefore, during the second sub-session, each subject received the other application in clear if the first application was obfuscated ($\mathcal{T}_{C,Lo} \leftrightarrow \mathcal{T}_{O,Ly}$ and $\mathcal{T}_{O,Lo} \leftrightarrow \mathcal{T}_{C,Ly}$). Table I summarises the four combinations of tasks assigned in the two sub-sessions. In addition, we paid attention to assign each task to the same proportion of subjects in the first - hence, also in the second - sub-session.

B. Variables

This section describes the variables used to perform the evaluation of the experiment. As *dependent variables*, we consider the following aspects of the executed attack tasks:

Correctness of a performed attack task. The correctness variable is evaluated as:

$$Corr(\mathcal{T}, s_i) = \begin{cases} 1 & \text{if subject } s_i \text{ succeeded in task } \mathcal{T} \\ 0 & \text{if subject } s_i \text{ failed in task } \mathcal{T} \end{cases}$$

Time to perform an attack task. The variable $Time(\mathcal{T}, s_i)$ is measured as the number of minutes spent by subject s_i to perform task \mathcal{T} , successfully or not.

Efficiency of an attack task. The efficiency variable, related to task \mathcal{T} , is the sum of the inverses of the time for all those subjects who successfully performed the task \mathcal{T} . Formally:

$$Eff_{\mathcal{T}} = \sum_{s_i \in S_{\mathcal{T}}} \frac{Corr(\mathcal{T}, s_i)}{Time(\mathcal{T}, s_i)}$$

where $S_{\mathcal{T}}$ is the set of subjects that performed the task \mathcal{T} . Note that, even if the sum ranges on all the subjects involved in task \mathcal{T} , the numerator function $Corr$ excludes from the computation each subject s_i who failed the task \mathcal{T} (i.e., $Corr(\mathcal{T}, s_i) = 0$).

We observe that the efficiency considers only the successful attacks. A measure of efficiency considering all cases would be equivalent to the inverse of $Time$. We analyzed that variable too but decided not to report it because it brings no additional insights. Note that all the dependent variables are related to the duration of the experiment, i.e., the quantity they measure is not independent from the time we have assigned to students

for the tasks. That is, the more the time of the task, the more the students that can correctly complete it, and the higher the average time and the efficiency. This factor has been considered during the design and does not affect the results of our study, which aims at assessing the effectiveness of VarMerge obfuscation by computing dependent variable on clear vs. obfuscated programs in equivalent conditions.

As *independent variables*, we consider the following ones:

Treatment applied to the source code, i.e., whether obfuscation was applied to the code or not. This is the main factor in our design.

Application used in a task; this can be used to understand how code complexity influences the time or the correctness of the attack task.

Lab is the order of the sub-sessions in the experiment; this is required to assess the learning across subsequent tasks, in terms of how the experience gained during the first assigned attack task influences the behaviour observed in the second task.

Experience of the subjects, in terms of number of years they have practiced C language programming.

C. Hypotheses

We can formulate the following null hypotheses to be tested:

- H_{01} : VarMerge source code data obfuscation has no effect on the correctness of an attack.
- H_{02} : VarMerge source code data obfuscation has no effect on the time to perform an attack.
- H_{03} : VarMerge source code data obfuscation has no effect on the efficiency of an attack.

D. Materials and procedures

In this section we detail the procedure followed and the material used during the experiments. To perform the tasks, subjects have been provided with a PC equipped with the Code::Blocks IDE running on Windows 7. We selected Code::Blocks because all the subjects were familiar with it, having used it in different courses during their Bachelor and Master Degrees. Before starting the experiment, the following materials were distributed to the participants:

- the experience questionnaire, used to acquire knowledge about the experience of the subjects in C programming and reverse engineering;
- a description of the program to be attacked;
- a zip archive containing a Code::Blocks project with the program to be attacked. We provided a working and tested Code::Blocks project to prevent subjects from losing time in creating the project and avoid problems in the compilation and execution of the programs due to misconfigurations of the project.

For tasks involving the Lottery program, the executable file of the server was given, along with a batch file which automated its execution; the server has been executed locally, in order to avoid network related problems; we did not provide the server source code nor tools to tamper with binaries, since in a MATE scenario, the attacker

does not have access to the server code of client-server applications;

- the description of the attack task the subjects were asked to perform.

Before performing the assigned task, participants had to fill in the experience questionnaire. The items in the questionnaire include:

- 1) the work experience as a professional programmer;
- 2) the overall experience in C programming, expressed in years;
- 3) how long participants have been using an IDE for C programming;
- 4) their experience in using a C debugger, in terms of actions they are able to perform with it:
 - add breakpoints;
 - execute the program stepwise;
 - inspect the call stack;
 - inspect the program variables;

After the questionnaire, subjects were asked to execute the given task, following the procedure below:

- 1) read a brief introduction that describes the program to be attacked;
- 2) install the Code::Blocks IDE, using an automated procedure made available on all PCs provided to subjects;
- 3) download the zip file, containing the Code::Blocks project from a given URL;
- 4) extract the Code::Blocks project from the archive file;
- 5) open the project with the Code::Blocks IDE;
- 6) (*for the Lottery program only*) start the batch file that runs the server;
- 7) build and execute the program (the client for the Lottery program);
- 8) (*for the Lottery program only*) look at the log file produced, trying to interpret the logged information;
- 9) read the description of the assigned task;
- 10) write down the task start time;
- 11) execute the task;
- 12) provide evidence that the task has been correctly executed:
 - for the Lotto program, write down the jackpot sequence;
 - for the Lottery program, show the assistants running the experiments the contents of the extractions accepted and logged by the server. As explained in Section III-A2, the server stores in this file all legal extractions received from the client, therefore assistants checked that the file contains only extractions with numbers between 1 and 20, so as to make sure that the subject has successfully completed the task;
- 13) write down the task completion time; we asked the students to not take into account the time elapsed for the initial setting of the experiments, and for eventual questions asked to the assistants; they therefore reported an estimation of the actual time spent solving the tasks.

Finally, participants had to fill in a post-experiment questionnaire, asking for their impressions on the task just completed. The questionnaire includes the following items:

- 1) whether the task was clear to the subject;
- 2) whether there was enough time to perform the task;
- 3) whether the subject felt that the task was easy to perform;
- 4) which tools the subject used to perform the task:
 - disassembler;
 - IDE;
 - debugger;
 - internet search;
 - other (open for the subject to specify);
- 5) on which activity most of the time was spent:
 - reading and understanding the binary;
 - inspecting the execution by means of the debugger;
 - changing the execution by means of the debugger.

The first three items were measured on a Likert scale with 5 levels.

E. Analysis method

The experimental measures are first summarised with basic descriptive statistics. Correctness is reported in terms of proportion of correct answers. For Time and Efficiency, we report the mean and standard deviation.

In all hypothesis testing we consider as independent variables the main factor – i.e. Treatment – and two co-factors, Application and Lab.

Among the two co-factors, Application is potentially the most relevant: in fact, we are considering two applications whose size (and possibly complexity) is quite diverse: a 238 LOC stand-alone vs. 62 LOC client-server application (in the Clear version). For this reason, we will report it in the summary tables and diagrams together with the main factor.

The Lab co-factor, representing the order of the lab task might be relevant in case maturation or learning effects emerge.

The two potentially confounding co-factors are included in all the hypothesis testing analyses. The motivation is that we aim at assessing the effect of the main factor once the co-factors have been accounted for.

To test hypothesis H_{01} , concerning Correctness, we use a logistic regression of Correctness vs. the three independent variables. Such analysis is suitable for the dichotomous nature of the measure. The logistic regression is based on the following model:

$$Correctness = \frac{1}{1 + e^{-(\beta_0 + \beta_T \cdot T + \beta_A \cdot A + \beta_L \cdot L)}}$$

where T and A are indicator variables for the Treatment and Application variables, while L indicates the assignment order. In particular:

$$T = \begin{cases} 1 & \text{if } Treatment = \text{Obfuscated} \\ 0 & \text{if } Treatment = \text{Clear} \end{cases}$$

$$A = \begin{cases} 1 & \text{if } Treatment = \text{Lotto} \\ 0 & \text{if } Treatment = \text{Lottery} \end{cases}$$

$$L = \begin{cases} 1 & \text{if first task} \\ 2 & \text{if second task} \end{cases}$$

To test hypotheses H_{02} and H_{03} concerning Time and Efficiency, we conduct a non-parametric test equivalent to ANOVA – permutation test – of the output variable vs. the three factors. The choice of a non-parametric test method is due to the expected non-normality of the measures. The linear regression is based on the following model:

$$V = \beta_0 + \beta_T \cdot T + \beta_A \cdot A + \beta_L \cdot L$$

Where V is the output variable (either Time or Efficiency) and the other variables are the same as those used in the logistic regression.

The assessment of the statistical test results is carried out assuming significance at a 95% confidence level ($\alpha=0.05$). Since we test three distinct hypotheses on the same set of participants, to avoid inflating the family-wise error rate we applied the Bonferroni correction; therefore we employ a corrected $\alpha_C = 0.05/3 = 0.017$ for decisions. So, we reject the null-hypotheses when $p\text{-value} < \alpha_C$.

All the data processing is performed with the R statistical package [14]. In particular the permutation test analysis was conducted using the `lmPerm` package [18].

IV. RESULTS

Before starting the analysis we looked for potential outliers. One subject qualified as such, showing an outlier timing (in excess) for the easiest task and an outlier timing (in defect) for the hardest task. This is probably due to a mistake in the annotation of the time. We decided to discard the subject from any further analysis.

Table II reports the descriptive statistics for the three output variables for different combination of Treatment and Application levels.

A. H_{01} : Correctness

The effect of the treatment on the Correctness is visible in the dot-plot reported in Figure 2a. The results of the tests on the logistic regression are reported in Table III. Based on the results from the tests, we cannot reject the null hypothesis H_{01} : the obfuscation Treatment has no significant effect on the Correctness of the attack task outcome. Only Application significantly affects the correctness of the task. The potential confounding factor Lab has no relevant effect.

TABLE II: Summary statistics for Correctness and Time.

Treatment	Application	N	Correctness		Time		Efficiency	
			n	prop.	mean	sd	mean	sd
Clear	Lottery	7	3	0.43	30.00	14.50	3.98	2.03
Clear	Lotto	6	6	1.00	6.50	3.08	10.75	4.05
Obfuscated	Lottery	7	1	0.14	94.57	14.37	0.65	NA
Obfuscated	Lotto	7	5	0.71	56.43	33.15	1.61	1.90

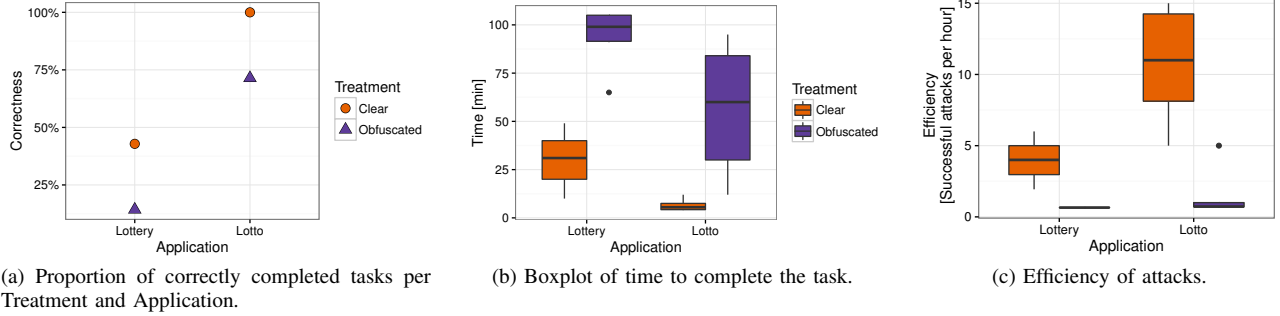


Fig. 2: Outcomes of the experiment

TABLE III: Logistic regression of Correctness

	Est.	Std.Err.	z value	Pr(> z)
β_0 (Intercept)	-0.101	1.991	-0.051	0.960
β_T TreatmentObfuscated	-2.001	1.240	-1.614	0.107
β_A ApplicationLotto	3.235	1.246	2.595	0.009
β_L Lab	-0.022	1.086	-0.020	0.984

TABLE IV: Permutation test of Time

	Estimate	Iter	Pr(Prob)
β_0 (Intercept)	35.566	5000	< 0.001
β_A ApplicationLotto	-32.799	5000	0.002
β_T TreatmentObfuscated	55.151	5000	< 0.001
β_L Lab	-9.511	316	0.241

B. H_{02} : Time

The distributions of Time for different combinations of Treatment and Application are reported in the boxplot of Figure 2b. The result of the permutation tests on the linear regression is reported in Table IV.

We reject the null hypothesis H_{02} : the obfuscation Treatment has a significant effect on the attack task time.

Similarly to what we observed for Correctness, also Time is affected by the specific Application considered in the task.

The potential confounding factor Lab has no significant effect.

The goodness of fit for the linear regression ($R^2 = 0.70$) is relatively high, considering the inherent difference among the participants.

C. H_{03} : Efficiency

The distributions of Time for different combination of Treatment and Application are reported in the boxplot of

TABLE V: Permutation test of Efficiency

	Estimate	Iter	Pr(Prob)
β_0 (Intercept)	4.786	5000	0.006
β_A ApplicationLotto	5.424	1233	0.109
β_T TreatmentObfuscated	-7.717	5000	0.002
β_L Lab	1.025	112	0.477

Figure 2c. The result of the permutation tests on the linear regression is reported in Table V.

On the basis of the tests, we can reject the null hypothesis H_{03} : the obfuscation Treatment has a significant effect on the attack Efficiency. Neither Application nor Lab have any effect on the Efficiency.

The goodness of fit for the linear regression ($R^2 = 0.59$) is relatively high, considering the inherent difference among the participants.

A practical measure of how much Obfuscation is effective in reducing the efficiency of an attacker can be computed by dividing the mean Efficiency on the clear code by that on obfuscated code, for either applications. The Efficiency ratios for the two applications are 6.1 for Lottery and 6.6 on Lotto.

D. Post-questionnaire

The post-questionnaire contains three items aimed at evaluating the perceived difficulties encountered by the participants while performing the attack tasks. The participant perception of clarity, availability of time, and easiness of the tasks is reported in Figure 3.

We observe that in general the task assignments were considered clear; somewhat less for the obfuscated Lottery.

Concerning the available time, when the participants worked on Lottery in a few cases they felt that not enough time was

allowed. No problem of time was reported when they worked on Lotto.

The different complexity of the two applications shows up in the responses to the third item concerning easiness of the task. The attack task on Lottery was considered more difficult. In addition, as expected, the tasks on obfuscated code turned out to be more difficult than those on clear code.

The second part of the post-questionnaire concerns the tools used to perform the task. Figure 4a shows the frequency of usage of each tool, by Application and Treatment.

We can observe two changes that occurred when the obfuscated version was used in the task: first, tools were used more; second, specifically the usage of the debugger increased. In addition, we observe different pattern of usage between the two applications. This reflects a difference that also emerged in the previous analyses.

Eventually, an item of the post-questionnaire addressed the activity on which participants spent most of their time during the task (see Figure 4b).

We can observe that the obfuscated versions of both applications required the participants to devote more time to execution of the application. This phenomenon appears in agreement with the increased usage of the debugger. Moreover, not surprisingly, we see a difference between the applications.

V. DISCUSSION

In the next sections we comment on the results and their validity.

A. Interpretation of results

The analysis results are summarized below.

- Data obfuscation significantly affects the time to complete an attack task (see Figure 2b) and the attack efficiency (see Figure 2c), while it does not affect the correctness of the attack outcome;
- The two applications significantly differ both in terms of time and correctness, while no difference can be observed for the efficiency. Such a disparity can be observed also in the perceived difficulties of the task (see Figure 3).
- The presence of obfuscation forces the attacker to resort more on all the available tools (see Figure 4a).
- Participants modified their attack strategy when facing obfuscated code: they mostly employed their time to execute the program. (see fig. 4b) The data on the tools used (see fig. 4a) allows us to infer that they attempted to understand the behaviour of the program by observing its dynamic evolution. This is in line with the anecdotal knowledge about data obfuscation in general and VarMerge in particular, which is expected to defeat attacks based on static analysis, while remaining relatively more vulnerable to dynamic analysis.
- By looking at the mean efficiency reported in Table II, we can observe a six-fold decrease in attack efficiency when the VarMerge obfuscation is used. Such effect is nearly the same for both applications.

We also observe that it is not possible to distinguish between subjects that did not succeeded in mounting the attack because they run out of time and the ones wouldn't have been able to actually perform the task. We highlight, however, that this distinction is irrelevant for our purpose of measuring the effect of VarMerge. Delaying successful attacks to the extent that they are no longer profitable is indeed one of the purposes of obfuscation techniques, which are not provable secure.

B. Threats to validity

We have checked our experiments against the checklist of the possible threats to validity proposed by Wohlin *et al.* [19], which are classified into construct, internal, conclusion, and external validity threats.

Construct validity threats concern the relationship between the theoretical constructs and the actual metrics defined for the experiment. While there are several types of attacks, we only focused on attacks that imply understanding and modifications of the source code. Alternative scenarios have not been tested. The presence of obfuscation is expected to affect both the comprehension and the change activities. However, in our experiment we could not measure the actual comprehension achieved by the subjects. Hence, we can only claim they achieved the minimum comprehension needed to perform their attack tasks, although we know they had to use more frequently sophisticated tools to understand the code when protections were applied. Time is one of the direct measures we collected; it is a coarse grained metric, including both comprehension and change. Though the two activities might be separate, a typical attack consists of a close interleaving of the two. The correctness of the attack task is evaluated as a boolean outcome. Although this is a very simple and crude metric, it reflects a real-case scenario where the attacker either gets access to the protected resources or not. Finally, all the threats related to mono-operation and method are excluded by design.

Internal validity is concerned with the capability to capture a cause-effect relation between the independent variables and the outcomes. That is, all noise factors, which may indirectly affect the outcomes, should have been eliminated or measured (e.g., assessed as negligible). Before starting any activity, the tasks and attack objectives have been explained to all subjects. The post-questionnaires confirmed that they had no problems in understanding the experiments. A maturation effect during the experimental session could have occurred: every subject has been assigned two tasks in sequence. Although we did not report it for the sake of readability, we tested the results for statistically significant effects of the order of the task. No significant effect was found. Since subjects were very homogeneous we could divide them randomly into groups. Finally, the experiment has been designed to avoid any ambiguity about direction of casual inference. Moreover, the inference stating that obfuscation renders attacks more complex did not appear ambiguous (and was already proved for analogous techniques in previous works).

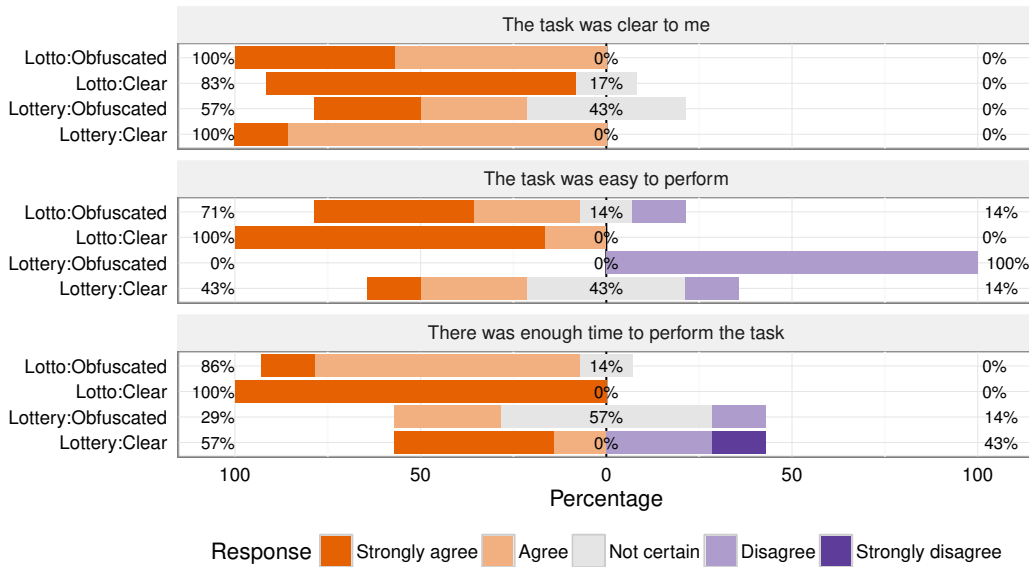


Fig. 3: Assessment of task difficulties

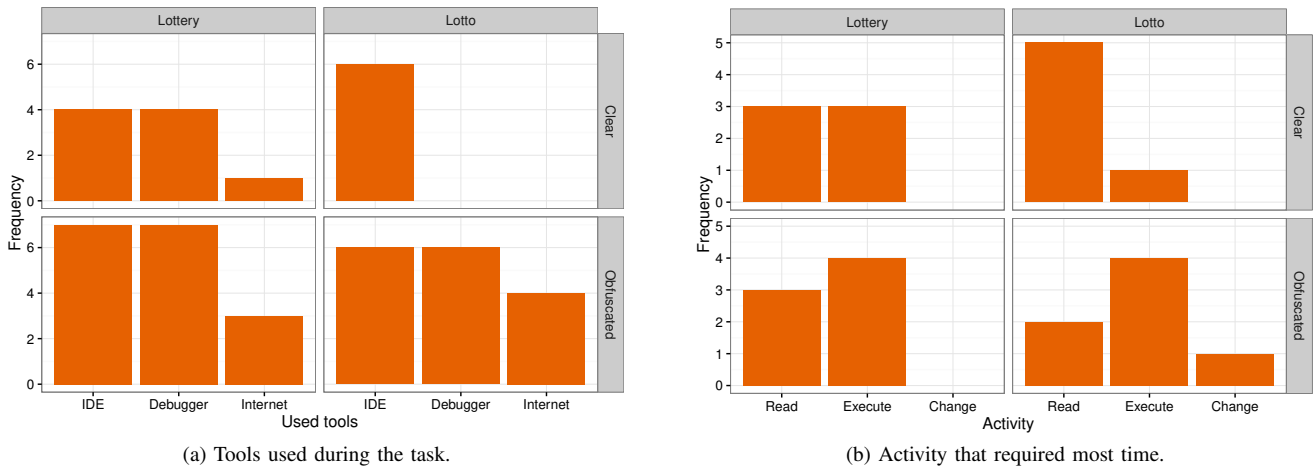


Fig. 4: Tools and activities analysis.

External validity threats are related to the impossibility to generalise our results to the case of real attackers who want to tamper with real applications protected by means of VarMerge. The selection of participants was made on a voluntary basis: greater motivation better represents realistic hackers' profile. While students expertise in hacking programs is far from that of "professional" hackers, the problem solving ability of the best students is not supposed to be very different from that of hackers. Given a fixed time frame, the expertise certainly affects the correctness variable in tampering with a given application, but the selection of the applications to protect has been fine tuned to have enough successful attacks in the 105 minutes available for the experiments even with students. Furthermore, we measure our outcomes and draw conclusions on the effectiveness of VarMerge by comparing the performance of subjects on clear and obfuscated versions of the application. Students with homogeneous expertise

give the same validity results as hackers with homogeneous expertise. This comparison mitigates the lack of expertise. Our experiment included two applications. However, we do not have enough findings to estimate how VarMerge may protect programs that are considerably different (e.g., larger or more complex) than the considered ones. This is one of the main directions for future work: including complexity metrics of the applications to protect as independent variables. However, complexity metrics have been not considered in this experiment as, given the size of our samples, we wouldn't have drawn any significant conclusion. Tools made available to subjects were up to date and valid representative of tools hackers may use. When we ran the experiment, no special event or news could have affected the data collected from subjects. Actually, we were not acquiring any subjective data, but their expertise and impression on the attack tasks. We are aware that the effectiveness variable we have introduced is

not the only way to measure the impact of successful attacks. Indeed, our formula favours attacks that are mounted quickly even if by a limited number of participants. We have decided for this approach as an application becomes vulnerable as soon as the first attacker succeeds. We have found less interesting alternative formulas that emphasize when more people succeeds in mounting an attack with higher average time. Moreover, it did not add any insight to what we already presented in Section III-E.

Conclusion validity threats are related to the validity of the methods to derive outcomes from the treatment data. We have used non-parametric statistical methods and controlled the error rate (permutation test ANOVA, error rate corrected with the Bonferroni method) as presented in Section III-E. We have collected data by means of survey questionnaires designed according to standard methods and scales [13]. Tasks were similar and balanced (one clear code and one obfuscated application; only VarMerge obfuscation used); subjects were not heterogeneous, as they were all master students, and experiments avoided random irrelevance.

VI. CONCLUSIONS

This paper reported an experiment aimed at assessing a specific data obfuscation technique – VarMerge – in terms of its capability to hinder and delay an attack. The experiment involved 15 students from the master degree programme in Computer Engineering at Politecnico di Torino. The experiment revealed no significant difference in terms of attack success rate between obfuscated and clear programs. This is mainly due to the size of the sample. In contrast, a significant difference was observed in the time required to complete the attack task. In addition, the results show that the presence of the VarMerge obfuscation is able to reduce by six times the attack efficiency (measured as the number of successful attacks per unit of time). This outcome provides a practical clue that can be used in designing software protections based on data obfuscation. However, since it is the first experiment that addresses data obfuscation techniques, we cannot draw conclusions on the comparison with other techniques. Moreover, we had no possibilities to apply other techniques as given the size of the sample.

In addition to the presence of obfuscation, the attack efficiency and time appear to be affected by the size and complexity of the program under consideration. This additional factor did not interfere, in our results, with the effect of obfuscation. Though this outcome is far from being conclusive due to the limited range in size and complexity that was investigated in our experiment.

As further work, we plan to:

- 1) test the VarMerge technique to other programs, with a wider variability in terms of size and complexity;
- 2) apply the same technique to binary programs;
- 3) apply and compare other obfuscation techniques;
- 4) involve subjects with different attacker skills.

Of course, considering all these independent variables and confounding factors needs proper preparation of the experiments and high number of participants, which we cannot reach in our institutions. Evaluating the effect of several obfuscation techniques on different applications can only be achieved with the collaboration and sharing of the results among researchers in the software engineering field. We have started investigating how to build such a community, prepare a planning of experiments and share data.

ACKNOWLEDGEMENT

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

REFERENCES

- [1] A. Akhunzada, M. Sookhak, N. B. Anuar, A. Gani, E. Ahmed, M. Shiraz, S. Furnell, A. Hayat, and M. K. Khan. Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. *J. Network and Computer Applications*, 48:44–57, 2015.
- [2] B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, and B. Preneel. Program obfuscation: a quantitative approach. In *Proc. ACM Workshop on Quality of protection*, pages 15–20, 2007.
- [3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im) possibility of obfuscating programs. *Lecture Notes in Computer Science*, 2139:19–23, 2001.
- [4] 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*, pages 1–39, 2014.
- [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*, 19(4):1040–1074, 2014.
- [6] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. The effectiveness of source code obfuscation: An experimental assessment. In *IEEE 17th International Conference on Program Comprehension (ICPC)*, pages 178–187, may 2009.
- [7] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition, 2009.
- [8] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Dept. of Computer Science, The Univ. of Auckland, 1997.
- [9] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski. Guest editors’ introduction: Software protection. *Software, IEEE*, 28(2):24–27, 2011.
- [10] J. Feigenspan, C. Kästner, J. Liebig, S. Ape I, and S. Hanenberg. Measuring programming experience. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 73–82. IEEE, 2012.
- [11] H. Goto, M. Mambo, K. Matsumura, and H. Shizuya. An approach to the objective and quantitative evaluation of tamper-resistant software. In *Third Int. Workshop on Information Security*, pages 82–96. Springer, 2000.
- [12] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. ACM Conf. Computer and Communications Security*, pages 290–299, 2003.
- [13] A. N. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter, London, 1992.
- [14] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.
- [15] I. Sutherland, G. E. Kalb, A. Blyth, and G. Mulley. An empirical examination of the reverse engineering process for binary files. *Computers & Security*, 25(3):221–228, 2006.
- [16] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 45–54, Washington, DC, USA, 2005. IEEE Computer Society.
- [17] C. A. Visaggio, G. A. Pagin, and G. Canfora. An empirical study of metric-based methods to detect obfuscated code. *International Journal of Security & Its Applications*, 7(2), 2013.

- [18] B. Wheeler. *lmPerm: Permutation tests for linear models*. R package version 2.0. Kluwer Academic Publishers, 2000.
- [19] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*.