Empirical Software Engineering (EMSE) manuscript No.
(will be inserted by the editor)

A Large Study on the Effect of Code Obfuscation on the Quality of Java Code

Mariano Ceccato, Andrea Capiluppi,
Paolo Falcarin, Cornelia Boldyreff

the date of receipt and acceptance should be inserted later

Abstract
Context: Obfuscation is a common technique used to protect software against malicious reverse engineering. Obfuscators manipulate the source code to make it harder to analyze and more difficult to understand for the attacker. Although different obfuscation algorithms and implementations are available, they have never been directly compared in a large scale study.
Aim: This paper aims at evaluating and quantifying the effect of several different obfuscation implementations (both open source and commercial), to help developers and project manager to decide which one could be adopted.
Method: In this study we applied 44 obfuscations to 18 subject applications covering a total of 4 millions lines of code. The effectiveness of these source code obfuscations has been measured using 10 code metrics, considering modularity, size and complexity of code.
Results: Results show that some of the considered obfuscations are effective in making code metrics change substantially from original to obfuscated code, although this change (called potency of the obfuscation) is different on different metrics. In the paper we recommend which obfuscations to select, given the security requirements of the software to be protected.

Mariano Ceccato
Fondazione Bruno Kessler–IRST, Trento, Italy
E-mail: ceccato@fbk.eu

Andrea Capiluppi
Brunel University, Kingston Lane, Uxbridge, London UB8 3PH
E-mail: andrea.capiluppi@brunel.ac.uk

Paolo Falcarin
University of East London, London E16 2RD, UK
E-mail: falcarin@uel.ac.uk

Cornelia Boldyreff
University of Greenwich, Park Row, London SE10 9LS
E-mail: c.boldyreff@gre.ac.uk

1 Introduction

Software protection is increasingly becoming an important requirement for industrial software development. Historically, software protection first appeared as attempts at adding license-checking code to computer games, followed by algorithms for white-box cryptography [30] used for digital media piracy protection. Every software vendor should be aware of the potential for man-at-the-end (MATE) attacks against their products and the techniques available to mitigate these attacks. MATE attacks take many forms: in a *tampering attack*, the user breaks the integrity of a piece of software, by modifying it in ways not intended by the software vendor. In a malicious *reverse engineering attack*, the attacker violates the confidentiality rights of the vendor by extracting intellectual property contained in the software, such as algorithms. Finally, in a *cloning attack*, copyright laws are violated by cracking and distributing illegal copies of the software. Methods for protecting against MATE attacks are variously known as software protection [12].

The software protection problem is fundamentally harder than other security problems. The reason is the wide attack model that software protection researchers and practitioners must compete with; one has to assume an almighty adversary who has full access to the chosen software and hardware and can examine, probe, and modify it at will. For this reason, no piece of software, however well protected, is expected to survive intact "in the wild" for a long period of time. An example of a very common form of protection against reverse engineering attacks is obfuscation which modifies a program to make it harder for the adversary to analyze or comprehend [11].

Code obfuscation was first discussed by Cohen [8] as a technique for automatically creating multiple versions of the same program, thereby making each version a more difficult target for malware to analyze and modify. Typical code obfuscation techniques [11] include splitting code into smaller pieces, merging pieces of unrelated code, randomizing code placement and instruction selection, breaking abstraction boundaries, mapping initially clean data structures to mangled ones, and flattening or introducing bogus control flow.

Essentially, obfuscation aims to work in the opposite direction to refactoring; code obfuscators should work against the understandability of the code. Given that metrics have been proposed to interpret and guide the refactoring effort, for instance in the presence of bad smells [24], the obfuscation tools should attempt to increase the same metrics that the refactoring tools are designed to decrease. For instance, the broad aim of refactoring is to "decrease the complexity of the code", hence the code obfuscators should provide algorithms to increase code complexity. This aspect and the fact that there are currently no practical security metrics to measure the quality of the protection poses the two following questions:

1. What is the effectiveness of different obfuscation algorithms?
2. What is impact of obfuscation on the code?
3. What protections should be chosen when securing a software system?

This paper studies the effect of 44 obfuscation algorithms contained in three obfuscation packages: the Sandmark tool [9], the Allatori suite and the Zelix Klassmaster^{TM}toolkit. Obfuscations are applied on 18 applications covering a total of 4 millions lines of Java code. The aim is to produce a set of obfuscated classes and to compare the algorithms in terms of code metrics (on modularity, size and complexity).

This paper is structured as follows. Section 2 discusses the related work in software protection metrics. Section 3 illustrates the obfuscation algorithms applied in this study and Section 4 presents the experimental design adopted in the study. Then Sections 5, 6 and 7 show the comparison results of the obfuscation algorithms. Section 8 discusses the implications and outlines the threats to validity, while Section 9 concludes the paper.

2 Related Work

With sufficient effort, most obfuscation techniques can be defeated. All of the information needed to break a software system is present in the executable or bytecode, and these can be controlled by the attacker. Assessing software protections means to estimate the extra delay the most sophisticated attacker would incur due to a particular protection technique used on a given application.

The evaluation of the increased strength introduced by obfuscation techniques has been mainly addressed by using code metrics [1, 13, 14, 16, 18, 20]. Even if metrics are based on reasonable assumptions about the expected problems that an attacker would face to defeat code protection, they just estimate and approximate a specific level of security that the underlying application should receive. Experimental evidence suggested that metrics correlate to the actual difficulty of performing attacks [4–6, 25].

Despite the benefits of experimental investigation, in the security literature only a few works are based on attacks performed by human subjects on binary code [25] and even fewer works [4,6] on applied empirical approaches, because they are expensive and time consuming. As an example, the comparison of just two obfuscation techniques and maintainability by users took a long time [5]. In this paper we adopt a different perspective; instead of comparing obfuscations in terms of how long attackers take to break them, we have measured the effects of the obfuscation by quantifying the changes occurred to source code in terms of modularity, complexity and size. In order to provide a replicable study, we have provided a statistically sound evaluation of several obfuscations with a wide set of metrics on a set of subject applications for a total of 4 millions lines of code analyzed.

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 (s)he tries to break the code. For example, Anckaert et al. [1] evaluate the obfuscation efficacy by using a specific set of metrics for control-flow and data-flow complexity. Linn et al. [20] 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.

A more high-level approach has been proposed by Collberg et al. [11] 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. In our research we use a similar approach when performing the correlation analysis of the variance of the chosen metrics.

More recently, Karnick et al. [18] 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).

Heffner and Collberg [14] 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. [16] presented a framework for iteratively combining and applying protection primitives to code; they also used code size, cyclomatic number and knot count metrics to evaluate the code complexity. Our work is in the same line of research, and brings the added value of replicating the experiment on a very large scale, and comparing 44 obfuscating algorithms.

Alternatively, tools have been used to assess the *resilience* of code obfuscations. For instance, Sutherland at al. [25] relied on a program binary instrumentation tool to measure the fraction of the obfuscating transformations that the attackers can undo automatically. Goto et al [13] proposed the *depth of parse trees* as a measure of source code complexity. Udupa et al [26] measured the resilience of an obfuscation by using the amount of time required to perform the automatic de-obfuscation to evaluate the effectiveness of control flow flattening obfuscation, relying on a combination of static and dynamic analysis. Our approach measures one obfuscation at a time. As a development of our research, it would be very relevant to study the application of a series of obfuscation algorithms on the same code, and analyze whether the resilience increases as compared to when a single obfuscation algorithm is applied.

Finally, Visaggio et al. [28] used the *code entropy* and size to detect obfuscated malware code in Javascript; their goal and metrics are different from ours but they also analyze the difference between metrics values obtained from obfuscated code and non-obfuscated code. Zeng et al.. [31] analyzed different obfuscations to discover which ones can break different types of watermarks hidden in the code.

3 Obfuscation Algorithms

In this study we compare the effects of several obfuscation transformations on Java code. We have selected three of the most prominent and used tools: the Sandmark obfuscating toolset [11][1], the Allatori Java obfuscating toolset[2] (version 4.1) and the Zelix Klassmaster™Java obfuscator[3] (version 5.5.0), for a total of 44 different atomic obfuscation algorithms.

The first (Sandmark) has been selected because it provides levels of flexibility, customization and openness that other obfuscation tools lack[4]; the second (Allatori) has been selected because it provides an all-in-one suite for obfuscating Java classes, and it provides a free toolkit; and the third (Klassmaster™) because it represents the state-of-the-art obfuscation toolkit, albeit at a licensing price. Below we detail how each tool has been used, together with the options that have been activated to produce the obfuscated outputs.

[1] `http://sandmark.cs.arizona.edu/downloads.html`

[2] `http://www.allatori.com/`

[3] `http://www.zelix.com/Klassmaster/`

[4] On the downside, Sandmark is quite old and it cannot handle the newest Java constructs, from Java version 1.5 onwards.

3.1 Sandmark

The Sandmark software tool can perform obfuscations, as well as statically and dynamically watermarking the source code and the binaries of Java systems. Being open-source, it also provides a full list of obfuscators, grouped in three categories: application-, class- and method-level. Method- and class-level obfuscations are more configurable as they allow developers to select which methods (or classes) to obfuscate; all other algorithms are considered application-level obfuscations. Such selections of partial obfuscation might be useful to prevent some methods from being obfuscated for the sake of performance and reliability; for example, obfuscation of reflective code can break the application (i.e. not preserving program semantics) while some obfuscations can penalize performance. Table 1, Table 2 and Table 3 summarize the characteristics of all the 40 algorithms used in this study, dividing them respectively in APP (application-), CL (class-) and MET (method-) level obfuscation algorithms. These descriptions are taken from the manual provided with the toolkit.

3.2 Allatori

The second obfuscation tool used, Allatori, can also be streamlined and activated via a command line interface. It provides methods to obfuscate and watermark the Java classes. The command line invocation requires an xml configuration file to activate the options for the obfuscation and watermarking. Since we were only interested in obfuscation, the options for the watermarking have been disabled. Browsing the documentation provided with the toolkit, we concluded that only 2 configurations are appropriate. Therefore two configurations have been created and analyzed, one with the control flow obfuscation activated (termed *cfo*) and another "light" configuration, without control-flow obfuscations, but with the renaming of the local variables (termed *lvo*), as summarised in Table 4.

3.3 Zelix Klassmaster^{TM}

The third obfuscation tool used, Zelix Klassmaster^{TM}, is also a commercial tool that provides several activation points for the obfuscation of the Java classes under study. It also provides a way to preserve methods, classes and packages from obfuscation, in order to focus very carefully the effects of the obfuscation algorithms. The tool can be streamlined by the use of scripts, which make it very powerful and automatisable.

Analyzing the documentation of the toolkit, it is evident that more control on the output obfuscation is given by Zelix than Allatori, albeit less than with Sandmark. Therefore, we have created two configuration files ("aggressive" and "light" obfuscation scenarios) which by activating switches can be used to affect the control flow attributes in more or less depth[6] as described in Table 5.

To try and understand more practically the meaning and effect of these switches, we tried different values of the same switch on a small Java class and we compared the results. Our observations are reported on the last column of Table 5.

[6] Most of these switches are self-explanatory, but `http://www.zelix.com/Klassmaster/docs/obfuscateStatement.html` provides a full description.

Name		Description
Array Folding	APP-af	takes a one-dimensional array and folds it into a multi-dimensional array.
Array Splitting	APP-as	takes a one-dimensional array field and splits it into 2 arrays by adding another field of the same type: one array will contain the first half of the elements and the other array will contain the second half
BLOAT	APP-bl	*BLOAT* is a Java bytecode optimizer performing many traditional program optimizations such as constant-copy propagation, constant folding, dead code elimination, and peephole optimizations [15].
Block Marker	APP-bm	randomly marks all basic bytecode blocks in the program with either 0 or 1, to be used to hide a watermark or slightly diversify the bytecode.
Class Encrypter	APP-ce	encrypts class files and causes them to be decrypted at runtime.
Constant Pool Reorder	APP-cpr	reorders the constants in the bytecode constant pool and assigns random indices to them: there is no change in code as a result of this obfuscation.
Dynamic Inliner	APP-di	inlines methods at runtime using *instanceof* checks.
False refactoring	APP-fr	it is performed on two classes that have no common behavior. If both classes have instance variables of the same type, these can be moved into a new parent class, whose methods can be buggy versions of some of the methods from the original classes.
Integer Array Splitting	APP-ias	splits a single array of integers into two arrays and based on some encoding method, the elements are put in either of the two arrays.
Interleave Methods	APP-im	finds pairs of methods in the input application and interleaves them into one method. It selects pairs such that both methods have the same signature and are not Java library's methods (e.g. toString()).
Overload Names	APP-on	obfuscates methods so that as many methods as possible have the same name. Method overriding relationships remain intact, whereas existing overloaded methods may be destroyed, and new ones created.
Parameter Alias	APP-pa	looks at each class and tries to find a (non-initializer, non-abstract, non-native) method that takes some object type as a parameter. It then aliases that parameter within the method using ThreadLocal class.
Rename Registers	APP-rr	renames local variables to random identifiers.
Split Classes	APP-sc	obfuscates a class file by splitting a node into two, i.e. some of the fields from the class are moved into a newly created class and all references to those fields in the given class are modified to reflect the changes.
String Encoder	APP-se	obfuscates the literal strings of a program. Each string is obfuscated and any string reference is replaced by a call to a method that de-obfuscates it.

Table 1 Sandmark – Description *application* level obfuscation algorithms (APP).

4 Experimental Setup

This section reports the definition, design and settings of the experiments in a structured way, following the template and guidelines by Wohlin *et al.* [29].

Name		Description
Class Splitter	CL-cs	adds several spurious classes by splitting the original, non-obfuscated ones into several obfuscated ones. It works at class-level, instead of at system level.
Field Assignment	CL-fa	obfuscates a class by inserting a bogus field into a class and then making assignments to that field in specific locations throughout the code. The specific locations are determined by the random selection of a "sibling" field.
Method Merger	CL-mm	merges all of the public static methods that have the same signature in each class into one large master method.
Objectify	CL-ob	takes a class and replaces all the fields with fields of the same name that have type *Object*; the algorithm runs through the entire application and fixes the proper references to the modified fields.
Publicize Fields	CL-pf	Makes the fields of a class public.
Simple Opaque Predicates	CL-sop	implements simple boolean identities and adds them to the code. Opaquely true constructs are embedded in the code, e.g. some constructs based on algebraic properties and known facts in mathematics.
Static Method Bodies	CL-smb	splits all of the non-static methods into a static helper method and a non-static stub that calls it.

Table 2 Sandmark – Description of *class* level obfuscation algorithms (CL).

4.1 Research Questions

The research questions that we intend to investigate in this study are the following:

RQ_1 What is the *potency* of code obfuscation?
RQ_2 Do different obfuscations have a different *impact* on source code?
RQ_3 Does *initial quality* of clear code influence the obfuscation potency?

4.2 Experimental Definition

The *goal* of this study is to analyze the effect of a source code obfuscation with the *purpose* of evaluating its effectiveness in defactoring source code. The *quality focus* regards how obfuscation impact the code with respect to complexity, modularity and size.

Results of this study can be interpreted from multiple *perspectives* as follows:

- a researcher interested to empirically assess obfuscation; and
- a software developer or project manager, who wants to ensure high resilience to attacks to subsystems of a sensitive applications running in an untrusted environment, before delivering it to the clients.

The *context* of the experiment consists of 18 Java subject applications, which have been subjected to code obfuscation.

Name		Description
Bludgeon Signatures	MET-bs	converts all methods to take Object[] parameter and return Object.
Boolean Splitter	MET-bsp	detects boolean variables and arrays and modifies their uses and definitions, by splitting each into 2.
Branch Inverter	MET-bi	exchanges the "if" and the "else" part of an if-else statement. It also negates the if condition so that the semantics is preserved.
Buggy code	MET-bc	selects a random method from the class file, and a random basic block in the method: a copy of the basic block is made and some additional bug codes are also introduced in this new basic block which changes the local variable values. This basic block is bypassed from execution.
Duplicate Registers	MET-dr	creates an additional variable that has its value changed according to an original local variable. Each reference to that variable value may have been changed to reference the new variable instead.
Inliner	MET-il	inlines static method bodies throughout the code replacing method invocations.
Insert Opaque Predicate	MET-iop	inserts an opaque predicate into every boolean expression. The boolean expressions are all relational operators that compare integers, so the opaque predicates will simply add an opaquely false value (i.e. value==0) to one of the integer operands.
Irreducibility	MET-ir	adds conditional branches to a method via opaque predicates so that the control flow graph of the resulting method is irreducible.
Merge Local Integers	MET-mli	combines two int variables into a single long variable, making access to either more confusing.
Opaque Branch Insertion	MET-obi	randomly inserts branches into a method.
Promotion Primitive Registers	MET-ppt	replaces all the local int variables in a function with local java.lang.Integer. This is possible through byte code manipulation of the all of the instructions that depend on retrieving and storing int values.
Primitive Promoter	MET-pp	changes all primitives in every method into instances of the respective wrapper classes.
Random Dead Code	MET-rdc	adds bogus statements onto the end of a Java method. The appended code may include a variety of other instructions including return instructions. Methods not ending in a *return* statements will impede reverse engineering tools.
Reorder Instructions	MET-ri	tries to reorder the instructions within each basic block of a method. The algorithm first creates a list of expression trees within each block. Once the dependency graph is obtained it writes out the instruction by doing a topological sort of the nodes in the dependency graph.
Parameter Reorderer	MET-pr	shuffles the argument orders for all methods.
Transparent Branch Insertion	MET-tbi	randomly inserts branches into a method. The branch will test to see if an Object field of the class is null, and if so it will branch.
Variable Reassigner	MET-vr	reallocates the local variables in a method, in order to minimize the number of local variable slots used.

Table 3 Sandmark – Description of *method* level obfuscation algorithms (MET).

4.3 Context: Subject Applications

In order to perform the experiment in realistic settings, we have considered the most active Java projects hosted in one of the largest Open Source portal (SourceForge[7]).

[7] `http://sourceforge.net`

Switch name	cfo	lvo	Description
control-flow-obfuscation	enabled	disabled	With this switch Allatori alters the control flow of the methods. The documentation explains that "it will not change the application behaviour at run-time, but will make the decompilation process much harder." [5]
variables renaming	disabled	enabled	With this switch Allatori renames the local variables found in the source code.

Table 4 Configuration of the two alternative versions of Allatori

A summary of the characteristics and the domains of the selected systems is available in Table 6. The applications vary both in terms of topic and size. At the time of writing (July 2012) 10 of them are among the 25 most downloaded and active projects on SourceForge[8]. The complete application set consists of 18 applications, including approximately 100k methods and almost 10k classes, giving a total of more than 4 millions lines of code.

4.4 Metrics

As stated in the research questions, different obfuscation techniques may have different impacts on the source code. For instance an obfuscation approach could focus on changing a specific aspect of the code (e.g., complexity) at the cost of overlooking others (e.g., modularity and size). As suggested by Collberg and Thomborson in their seminal work [10], we consider several software engineering code metrics to quantify the measurable effect of code obfuscation, in terms of the complexity, the modularity and the size of obfuscated code.

Complexity Metrics Code complexity is measured using the McCabe cyclomatic complexity index [22]. Even though this metric was initially proposed for procedural code (e.g., C, ADA, etc.), its adoption in object oriented languages has been often discussed and partially validated by other authors [21, 27].

As this metric is computed on the source code, we need to decompile obfuscated code. To achieve this aim, we use the *jad* decompiler [19], and the resultant source code has been analyzed with *scitool*[9] in to compute the McCabe cyclomatic complexity.

Modularity Metrics The Chidamber and Kemerer (C&K) object-oriented metrics framework [7] is a suite of metrics for OO design, and is composed of the following modularity metrics:

- Weighted Methods Per Class (WMC),
- Depth of Inheritance Tree (DIT),
- Number of Children (NOC),
- Coupling Between Object Classes (CBO),
- Response For a Class (RFC), and
- Lack of Cohesion in Methods (LCOM).

[8] As provided in the "Recently updated" section of the Java applications, `http://sourceforge.net/directory/language:java/os:linux/freshness:recently-updated/`.

[9] `http://www.scitools.com/`

Switch name	Aggressive	Light	Description
aggressive-Method-Renaming	true	false	This option has an effect only on the way in which the names are produced in the obfuscated code. The level of such aggressive renaming can be either "true" or "false".
keepInnerClass-Info	false	true	This option has an effect on whether Klassmaster keeps track of inner classes (i.e., the switch is "true") or not (i.e., the switch is "false"). This information is maintained in case the names of the inner classes have not been obfuscated (i.e., the switch is "ifNameNotObfuscated").
keepGenerics-Info	false	true	As for inner classes, Klassmaster can keep track of generics (i.e., the switch is "true") or not (i.e., the switch is "false").
obfuscateFlow	aggressive	light	Klassmaster allows an obfuscation to act on the control flow. Its levels can be disabled (i.e. "none"), or having an increasing amount of obfuscation on control flow related statements, such selection constructs (if...else) and loop constructs (while, for):"light", "normal" and "aggressive" are the levels provided in this switch.
encryptString-Literals	aggressive	light	This option allows the obfuscation of string literals, and at various levels of obfuscation, from "none", to "normal", "aggressive" and "flowObfuscate" (i.e., using a flow obfuscated decrypt method).
exception-Obfuscation	heavy	light	This option perform a flow obfuscation which involves exceptions: by selecting "heavy" or "light", the obfuscation will be more or less aggressive.
autoReflection-Handling	normal	none	With this option, Klassmaster can handle the Java Reflection API calls which access classes, fields or methods. This switch can be off (i.e, "none"), or on (i.e., "normal").
lineNumbers	scramble	scramble	With this option, Klassmaster can maintain (i.e., the switch is "keep"), erase (i.e., "delete") and mix-up (i.e., the switch is "scramble") the map of bytecode instructions to source code line numbers.
localVariables	delete	delete	With this option, Klassmaster can maintain (i.e., the switch is "keep") or erase (i.e., "delete") the local variable tables in the bytecode that store the local variable names in the source code.
randomize	true	false	With this option, KlassMaster can generate (i.e., the switch is "true") or not (i.e., the swith is "false ") new obfuscated names for methods and classes in a random fashion.
allClasses-Opened	true	true	When this option is set to "true", it means that all the classes have been opened for obfuscation. If the option is "false", a mapping of all the unopened classes has to be provided.
derive-Groupings-FromInput-ChangeLog	false	false	When set to "false", KlassMaster automatically determines how to group classes to obfuscate the original structure.

Table 5 Configuration of the two alternative versions of Klassmaster^{TM}.

These metrics have been used extensively by researchers in the past few years, and they have been validated for OO languages [2].

The C&K metrics are computed by the *ckjm* tool[10] [17]. *Ckjm* calculates the C&K metrics by processing directly the bytecode of compiled Java files. This tool also calcu-

[10] Available at `http://www.spinellis.gr/sw/ckjm/`

System	Methods	Classes	LoC
CarC	214	26	1,712
Azureus2	36,578	3,252	1,163,809
Carserver	52	30	766
ChatC	144	26	1438
Chatserver	57	7	1,665
Freemind	4,804	405	129,394
GCS	2,468	199	68,060
Hfsx	2,843	344	144,599
Im4java	1,614	86	62,582
Ipscan	6,901	422	215,728
Jboss	7,679	710	244,461
Jml	1,893	236	39,395
Lwjgl	8,145	456	159,482
SQuirrel	5,410	395	101,347
SweetHome3D	3,812	177	151,144
Triplea	18,939	1,699	679,057
Tuxguitar	4,549	477	148,720
Weka	16,417	1,038	752,652
Total	**122,519**	**9,985**	**4,066,011**

Table 6 Summary of subject applications.

lates (for each class) the *Number of Afferent Couplings* (Ca), and the *Number of Public Methods* (NPM).

Size Metrics The size of the code is determined by counting the *Lines of Code*. Lines are counted using the linux *wc* utility on the java code obtained after decompiling the bytecode.

Potency Eventually, we quantify the effect of obfuscation by comparing complexity, modularity and size before and after obfuscation. To achieve this aim we adopt the notion of obfuscation *Potency*, originally proposed by Collberg and Thomborson [10]. For each metric M Let $M(P)$ be the complexity of the clear program P, and $M(P')$ the complexity of the program P' obfuscated with the transformation T. Potency of T with respect to the program P and metric M is defined as:

$$T_{pot}(P, M) \stackrel{\text{def}}{=} \frac{M(P')}{M(P)} - 1 \tag{1}$$

However, metrics are computed per class and not per program, so metrics do not provide a single value for a program but a set of values. In order to turn potency into an operative metric, we have to rely on the average values $\overline{M}$ of the metric M computed on all the classes of the program P and P'. The operative definition of potency used in our study is:

$$T_{pot}(P, M) \stackrel{\text{def}}{=} \frac{\overline{M}(P')}{\overline{M}(P)} - 1 \tag{2}$$

4.5 Experimental Procedure

An overview of the toolchain adopted to prepare the code for the experiment is shown in Figure 1. First of all, the three obfuscation tools are applied to the original code, in order to obtain the obfuscated version. As mentioned above, the three obfuscation tools produce bytecode as an output. While modularization metrics can be computed directly on the bytecode, in order to measure complexity and size, it is necessary to decompile the classes to obtain Java source code. Invocations to the *jad* decompiler are piped after the production of the obfuscated bytecode, and the resultant source code is analyzed.

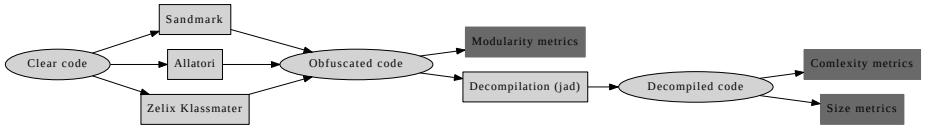

Fig. 1 Overview of the toolchain used.

4.6 Sanity Check

In the real world software systems may vary considerably in terms of code quality metrics. Before using the subject applications in our study, we have to perform a sanity check on the subject applications to exclude the possibility that they are too similar each other and bias the study. For the the purpose of the sanity check, we formulate the subsequent null hypothesis:

H_{0s} : There is no difference among applications in terms of the quality of source code;

To understand if the considered subject applications convey an adequate diversification, first of all we have measured their code qualities (i.e., complexity, modularity and size) and then we have studied their variance.

To achieve this objective, i.e., the test of hypotheses H_{0s}, we use the Analysis of Variance (ANOVA). Although ANOVA is a parametric test, it is considered quite robust also for non-normal and non-interval scale variables.

For each metric, we compute the ANOVA table of metric by Subject Application, and we report the results in Table 7 (p-values only). With the only exception of *Number of Children* (*noc*), for all other metrics we observe statistical significance. In these cases, we can reject the null hypothesis H_{0s} and formulate the subsequent alternative hypotheses:

- Subject applications are different in terms of (McCabe cyclomatic) *complexity*;
- Subject applications are different in terms of *modularity* (with respect to *wmc*, *dit*, *cbo*, *rfc*, *lcom*, *ca* and *npm*);
- Subject applications are different in terms of *size* (with respect to *LoC*);

Metric	P-value
McCabe	**<0.01**
wmc	**<0.01**
dit	**<0.01**
noc	0.05
cbo	**<0.01**
rfc	**<0.01**
lcom	**0.04**
ca	**<0.01**
npm	**<0.01**
LoC	**<0.01**

Table 7 Analysis of variance of subject applications by metric.

Thus, applications are quite diversified[11] in terms of complexity, modularity and size, so they represent an adequate set of subject applications to study the effect of code obfuscation.

5 RQ_1: Obfuscation Potency

This section reports on the analysis on obfuscation potency, i.e. the difference between clear and obfuscated code with respect to code metrics. For the first research question we formulate the subsequent null hypotheses:

H_{01c} : There is no difference in the *complexity* of clear and obfuscated code.
H_{01m} : There is no difference in the *modularity* of clear and obfuscated code.
H_{01s} : There is no difference in the *size* of classes on clear and obfuscated code.

The three null hypotheses are *two-tailed*, because we are interested in analyzing the effect of obfuscation in both directions, i.e. its increase and reduction. In fact, different obfuscation techniques may have different impacts on the source code. For instance, an approach could focus on changing a specific aspect of the code (e.g., complexity) at the cost of overlooking others (e.g., modularity and size). In case the analysis allows us to reject a null hypothesis, an alternative hypothesis will be formulated.

5.1 Overall Analysis

First of all, we measure the complexity, modularity and size of the classes from clear code. Then, we apply each obfuscation to each case study application, thus obtaining 44 versions of the 4 Mloc (one version per obfuscation). Finally, we measure the same metrics on obfuscated classes.

Before approaching a detailed analysis, we mean to perform an overall analysis, to see if the considered metrics capture any difference among all the treatments (i.e., obfuscations). Thus, we use the ANOVA test. This test can be used to decide whether to reject the null hypothesis that the distribution of a given the metric (e.g., Lines of Code) is the same across treatments (i.e., all the obfuscated code and clear code).

[11] Detailed analysis not reported for reason of space shows that the majority of them are different each other.

A distinct one-way ANOVA test is run for each metric, thus AOVA is run 10 times. Table 8 reports the Analysis of Variance of Metric by Treatment, a different metric per line. Statistical significance is assumed when the p-value is <0.05 (we assume significance at a 95% confidence level, α=0.05), significant cases are highlighted in boldface. As can be seen in the table, for almost all the metrics we can reject the null hypothesis of no difference, with the only exception of noc. So we can formulate the alternative hypothesis that there is some difference between obfuscated and clear code with respect to all the remaining metrics.

Relation tested	p.value
McCabe	**<0.01**
wmc	**<0.01**
dit	**<0.01**
noc	0.84
cbo	**<0.01**
rfc	**<0.01**
lcom	**<0.01**
ca	**<0.01**
npm	**<0.01**
LoC	**<0.01**

Table 8 Analysis of variance of Metric by Treatment.

Based on the experimental data, we can reject the null hypotheses H_{01c}, H_{01m} and H_{01s}. Thus, we can formulate the subsequent alternative hypotheses:

- The *complexity* of obfuscated code is different than complexity of clear code, and the magnitude of this difference is expressed by the *potency*[12] shown in Table 9.
- The *modularity* of obfuscated code is different than modularity of clear code, with respect to wmc, dit, cbo, rfc, lcom, ca and npm. The magnitude of this difference is expressed by the *potency* shown in Table 10.
- The *size* of obfuscated classes is different than size of classes from clear code, and the magnitude of this difference is expressed by the *potency* shown in Table 11.

In the rest of this section, post-hoc tests will be used to determine which obfuscation algorithms do in fact improve or degrade obfuscated code compared to clear code.

5.2 Analysis of Complexity

We start by considering the McCabe cyclomatic complexity of clear code and obfuscated code. Figure 2 shows the boxplots of the McCabe cyclomatic complexity. Obfuscations are sorted by ascending order of the average McCabe complexity (smaller averages on the left-hand side, higher averages on the right-hand side). The reference value of clear code is highlighted in red. By visual inspection, we can note that most obfuscations report a complexity similar to the clear code, i.e. with a mean of 1. Only few cases show higher values of complexity, on the right hand side of the plot.

[12] As suggested by Collberg [10], we use the *potency* to measure the magnitude of the difference of a specific metric between clear and obfuscated code.

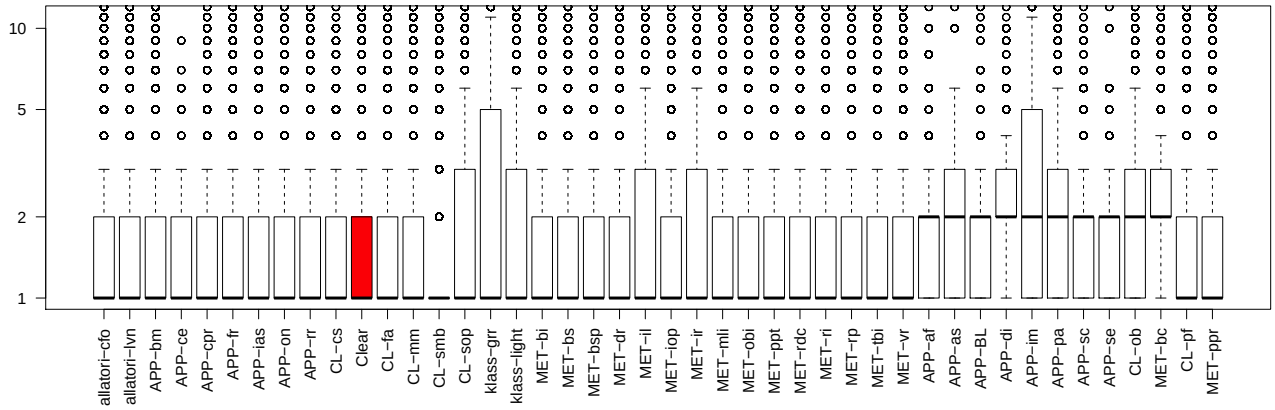

Fig. 2 Boxplot of McCabe cyclomatic complexity.

To see if the observations formulated in the graph are statistically significant, we compare the complexity of the clear code and the obfuscated code with the unpaired (two-tailed) Mann-Whitney test [23]. This test is non-parametric, so it does not make any assumptions on the normal distribution of the experimental data. Given the experimental settings, it is not possible to use a paired test. In fact some obfuscations rename or split classes and methods in a way that makes it hard to map obfuscated methods back to the original ones.

When multiple pairwise comparisons are performed with overlapping data, however, the number of hypotheses in a test increases and so does the likelihood of witnessing a rare event. Hence, the chance to reject true null hypotheses may also increase (type I error). To control this problem, we adopt the *Holm* correction which is more complex but also more powerful than the *Bonferroni* correction. The *Holm* correction consists of using different significance levels on different tests. *P*-values from the n dependent hypotheses are sorted in ascending order. Then, on each ordered $p\text{-}value_i$, a decreasing correction factor $n - i + 1$ is used, i.e., an increasing significance level $\alpha/(n - i + 1)$. We reject the null hypotheses until the minimum index k for which the null hypothesis cannot be rejected is encountered ($p\text{-}value_k > \alpha/(n-k+1)$). All subsequent hypotheses cannot be rejected ($p\text{-}value_i : i > k$).

While the statistical test allows for checking the presence of significant differences, it does not provide any information about the magnitude of such a difference. This is particularly relevant in our study, since we are interested to investigate to what extent the use of obfuscation changes the complexity of source code. To this aim we adopt the notion of obfuscation *Potency*, as defined in Equation (2).

For each obfuscation, Table 9 reports descriptive statistics (mean and standard deviation) of the complexity of clear code and obfuscated code (second and third columns), the p-value resulting by the and Mann-Whitney test and the potency of the obfuscation (fourth and fifth columns). Values of the original clear code are reported on the first line (mean McCabe complexity = 2.54). For those cases where difference in the McCabe cyclomatic complexity of clear code and obfuscated code is

Obfuscation	mean	sd	p.value	Potency
Clear	2.54	4.90		
allatori-cfo	2.42	4.60	<**0.01**	-0.05
allatori-lvn	2.41	4.58	<**0.01**	-0.05
klass-grr	4.89	12.14	<**0.01**	**0.93**
klass-light	3.32	7.15	<**0.01**	0.31
APP-bm	2.53	4.87	0.09	
APP-ce	1.93	1.47	0.03	
APP-fr	2.53	4.87	0.09	
APP-rr	2.53	4.87	0.09	
APP-af	2.45	2.45	<**0.01**	-0.04
APP-as	2.33	2.01	<**0.01**	-0.08
APP-BL	2.19	1.91	<**0.01**	-0.14
APP-cpr	2.13	2.79	<**0.01**	-0.16
APP-di	3.15	3.54	<**0.01**	0.24
APP-ias	2.55	4.09	<**0.01**	0.01
APP-im	4.27	6.11	<**0.01**	0.68
APP-on	2.43	5.75	<**0.01**	-0.04
APP-pa	2.75	4.31	<**0.01**	0.09
APP-sc	2.38	2.80	<**0.01**	-0.06
APP-se	2.25	2.08	<**0.01**	-0.11
CL-cs	1.92	2.47	<**0.01**	-0.24
CL-ob	2.84	4.45	<**0.01**	0.12
CL-sop	3.61	8.00	<**0.01**	0.42
CL-fa	2.53	4.87	0.09	
CL-mm	2.54	4.90	0.09	
CL-pf	2.52	4.85	0.35	
CL-smb	1.90	3.73	<**0.01**	-0.25
MET-bs	2.42	5.60	<**0.01**	-0.05
MET-bsp	2.13	2.79	<**0.01**	-0.16
MET-il	2.76	4.04	<**0.01**	0.09
MET-iop	2.14	2.80	<**0.01**	-0.16
MET-ir	2.45	3.08	<**0.01**	-0.03
MET-ppt	2.42	5.60	<**0.01**	-0.05
MET-ri	2.13	2.79	<**0.01**	-0.16
MET-rp	2.44	5.62	<**0.01**	-0.04
MET-vr	2.46	3.95	<**0.01**	-0.03
MET-bc	3.04	2.89	<**0.01**	0.20
MET-bi	2.38	4.49	<**0.01**	-0.06
MET-dr	2.53	4.84	0.06	
MET-mli	2.53	4.84	0.06	
MET-obi	2.97	7.14	<**0.01**	0.17
MET-ppr	2.51	4.79	0.43	
MET-rdc	2.53	4.84	0.06	
MET-tbi	2.64	5.10	<**0.01**	0.04

Table 9 Mann-Whitney test of McCabe cyclomatic complexity (with Holm correction) and Potency.

statistically significant, p-values are highlighted in boldface (Holm correction is used). To avoiding misleading interpretations, potency is reported only on those cases where statistical significance is observed. Moreover, the highest potencies are highlighted in boldface and the lowest potencies are underlined, to evidence those obfuscations that reported the most relevant change in the code.

While *allatori* reduced the McCabe complexity of the obfuscated code with respect to clear code (potency<0), *KlassMater* increased it considerably.

Among the application level obfuscations, the code obfuscated with *Array Folding* (APP-af), *Array Splitting* (APP-as), *Bloat* (APP-BL), *Constant Pool Reorder* (APP-cpr), *Overload Names* (APP-on), *Split Classes* (APP-sc), *String Encoder* (APP-se) achieves a significant reduction of the complexity (potency<0), while *Dynamic Inliner* (APP-di), *Integer Array Splitting* (APP-ias), *Interleave Methods* (APP-im), *Parameter Alias* (APP-pa) make complexity increase. The other application level obfuscations do not achieve statistical significance for McCabe complexity. Among class level obfuscations there are two cases that make complexity decrease (*Class Splitter* and *Static Method Bodies*), while just two increase complexity (*Objectify*, *Simple Opaque Predicates*). Eventually, when considering method level obfuscations, many significant cases are reported that make complexity either increase or decrease.

5.3 Analysis of Modularity

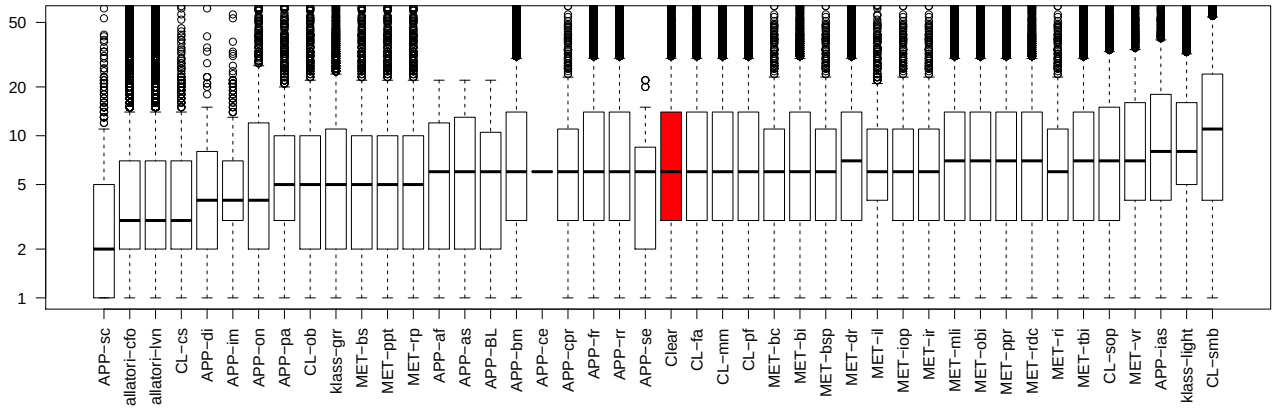

Fig. 3 Boxplot of Wighted Method per Class.

In a similar way we have analyzed the metrics related to modularity. Figure 3 shows the boxplot of the *wmc* metric (Weighted Method per Class) for the considered obfuscations. As in the previous case, the values for the original, non-obfuscated clear code are in red. From a visual inspection, we see that the clear code is in the middle of the spectrum and nearly half of the obfuscations make wmc decrease, while in other half of the case wmc increases with respect to clear code. Unlike from the previous case, the modularity values are not always similar to the case of clear code. For reason of space, we omit the boxplots of the other modularity metrics, but we present the results of the statistical analysis for all the metrics. All the boxplots are available in a technical report [3].

Table 10 reports the results of the statistical analysis for all the modularity metrics. To achieve a compact representation, the table does not report all the p-values (with the Holm correction). Only in the statistically relevant cases the table does report the

Obfuscation	Potency							
	wmc	dit	noc	cbo	rfc	lcom	ca	npm
allatori-cfo	-0.40	-0.03	-0.15		-0.35	-0.52		-0.42
allatori-lvn	-0.40	-0.03	-0.15		-0.35	-0.52		-0.42
klass-grr	-0.19		-0.11		-0.15	-0.76		-0.23
klass-light	0.09			0.15	0.09	-0.62	**0.14**	0.01
APP-bm								
APP-ce		**1.02**		-1.00			-1.00	
APP-fr								
APP-rr								
APP-af		0.59		-0.73		-0.96		
APP-as		0.58		-0.73		-0.97		
APP-BL		0.60		-0.73		-0.97		
APP-cpr		-0.09		0.24	0.03			
APP-di	0.21			-0.57		-0.97		
APP-ias	0.14	0.10	0.28	-0.06	0.18	-0.57	-0.03	**0.28**
APP-im	-0.34	-0.44			-0.37	-0.93		
APP-on	0.37			-0.41	-0.12	9.43	-0.28	0.24
APP-pa	0.05	-0.28		0.09		2.98		0.07
APP-sc	-0.50			-0.49	-0.59	0.18	-0.39	-0.34
APP-se		0.56		-0.69		-0.97		
CL-cs	-0.42	-0.30	**1.09**	-0.37	-0.41	-0.84	-0.33	-0.22
CL-ob	0.23	-0.27		-0.38	-0.16	**3.99**	-0.25	0.14
CL-sop	0.03				0.11	-0.52	0.01	0.05
CL-fa								
CL-mm								
CL-pf								
CL-smb	**0.78**				**0.27**	2.78		
MET-bs	0.23	-0.27		-0.40	-0.16	**3.99**	-0.28	0.14
MET-bsp		-0.09		0.24	0.03			
MET-il		-0.14		0.25	0.14			0.22
MET-iop		-0.09		0.24	0.03			
MET-ir		-0.09		0.24	0.03			
MET-ppt	0.23	-0.27		-0.38	-0.09	**3.99**	-0.25	0.14
MET-ri		-0.09		0.24	0.03			
MET-rp	0.23	-0.27		-0.38	-0.16	**3.99**	-0.25	0.14
MET-vr	0.10	0.07		0.09	0.19	-0.53		0.16
MET-bc		-0.11		**0.27**	0.04			
MET-bi								
MET-dr								
MET-mli								
MET-obi								
MET-ppr					0.05			
MET-rdc								
MET-tbi								

Table 10 Mann-Whitney test of Chidamber and Kemerer object-oriented metrics (with Holm correction) and Potency.

potency of the obfuscation in the line, with respect to the metric in the column. For each metric, we highlighted in boldface the highest and the lowest values of potency.

Inspecting the table row-wise, it is interesting to note that, when statistically significant, the potency of *allatori-cfo*, *allatori-lvn*, *klass-grr* is always negative, while the other obfuscations report positive potency on some metrics and negative potency on other metrics. Looking at the table column-wise, there is no consistent trend amongst

the metrics. In fact, for each metric, there are obfuscations that report a positive potency, and obfuscations that report a negative potency.

With some obfuscations, the obfuscated code is not significantly different from the clear code with respect to any modularity metrics. These obfuscations are *Block Marker* (APP-bm), *False refactoring* (APP-fr), *Rename Registers* (APP-rr), *Field Assignment* (CL-fa), *Method Merger* (CL-mm), *Publicize Fields* (CL-pf), *Branch Inverter* (MET-bi), *Duplicate Registers* (MET-dr), *Merge Local Integers* (MET-mli), *Opaque Branch Insertion* (MET-obi), *Random Dead Code* (MET-rdc), *Transparent Branch Insertion* (MET-tbi).

5.4 Analysis of Class Size

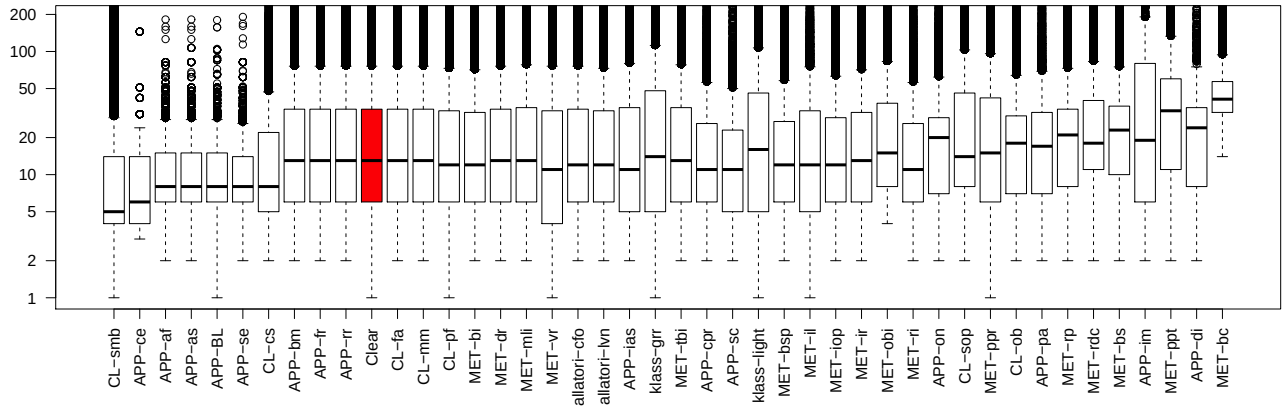

Fig. 4 Boxplot of Lines of Code.

Eventually, we analyze how obfuscation affects the size of the code. Figure 4 shows the boxplots of *Lines of Code* case studies applications. While only few cases make this metric decrease, a relevant number of obfuscations makes the *Lines of Code* increase on the right-hand side of the graph.

As can be seen in Table 11, in many cases the size is significantly different from that of the clear code. The table reports the mean and standard deviation of *Lines of Code* for clear code and obfuscated code, together with the p-value of the Mann-Whitney test and the potency. Only few cases are not statistically significant. For the significant cases, potency is sometime positive (size increases), sometime negative (size decreases).

Considering the observed results, additional research hypotheses can be formulated. First of all, some of the obfuscations considered in this study include non-deterministic decisions on how to obfuscate the code. For instance, there could be multiple ways of generating opaque predicates or splitting classes, and the obfuscator may rely on a random number generator to decide which alternative approach to take. We are interested in studying if non-deterministic behaviour influences these metrics.

Obfuscation	mean	sd	p.value	Potency
Clear	33.19	122.29		
allatori-cfo	33.60	118.47	<**0.01**	0.01
allatori-lvn	32.68	114.79	<**0.01**	-0.02
klass-grr	48.42	289.44	<**0.01**	0.46
klass-light	39.32	139.58	<**0.01**	0.18
APP-bm	33.34	121.78	<**0.01**	0.00
APP-ce	12.03	17.57	<**0.01**	-0.64
APP-fr	33.33	121.76	<**0.01**	0.00
APP-rr	33.33	121.76	<**0.01**	0.00
APP-af	13.72	14.10	<**0.01**	-0.59
APP-as	13.40	13.47	0.01	
APP-BL	13.39	14.08	<**0.01**	-0.60
APP-cpr	26.99	78.57	<**0.01**	-0.19
APP-di	29.59	39.38	<**0.01**	-0.11
APP-ias	35.38	88.29	<**0.01**	0.07
APP-im	59.05	122.22	<**0.01**	**0.78**
APP-on	24.29	71.29	<**0.01**	-0.27
APP-pa	30.74	82.52	<**0.01**	-0.07
APP-sc	20.31	30.80	0.13	
APP-se	13.17	13.74	<**0.01**	-0.60
CL-cs	18.85	42.61	<**0.01**	-0.43
CL-ob	28.43	73.10	<**0.01**	-0.14
CL-sop	48.20	129.58	<**0.01**	0.45
CL-fa	33.33	121.76	<**0.01**	0.00
CL-mm	33.39	121.97	<**0.01**	0.01
CL-pf	32.84	120.98	<**0.01**	-0.01
CL-smb	18.94	84.76	<**0.01**	-0.43
MET-bs	28.79	80.77	<**0.01**	-0.13
MET-bsp	27.68	79.22	<**0.01**	-0.17
MET-il	36.50	153.48	<**0.01**	0.10
MET-iop	32.82	92.36	<**0.01**	-0.01
MET-ir	31.36	82.02	<**0.01**	-0.05
MET-ppt	44.60	119.93	<**0.01**	0.34
MET-ri	26.99	78.57	<**0.01**	-0.19
MET-rp	26.90	74.00	0.03	
MET-vr	33.59	92.06	0.68	
MET-bc	53.52	82.72	<**0.01**	0.61
MET-bi	30.64	116.75	<**0.01**	-0.08
MET-dr	32.91	121.22	<**0.01**	-0.01
MET-mli	35.99	130.19	<**0.01**	0.08
MET-obi	45.37	162.61	<**0.01**	0.37
MET-ppr	39.48	149.55	<**0.01**	0.19
MET-rdc	37.28	121.54	<**0.01**	0.12
MET-tbi	33.43	103.35	<**0.01**	0.01

Table 11 Mann-Whitney test of Lines of Code (with Holm correction) and Potency.

Secondarily, in this study Zelix Klassmaster and Allatory have been involved that applied a combination of obfuscations, while Sandmark has been used only with individual obfuscations separately. So we are interested in studying if, when combining many obfuscations obfuscations together, the effect of the single obfuscations reinforce each other in the combination. So, we formulate these additional null hypotheses:

H_{01nd} : The *non determinism* of code obfuscation transformations does not influence complexity, modularity and size of obfuscated code.

H_{01r} : When applying different obfuscations in combination, different obfuscations do not *reinforce* each other.

5.5 Analysis of Non-deterministic Behaviour

In this subsection we study the impact of non-determinism on obfuscation potency. Among the case studies we have selected CarRace because no obfuscator fails in (i.e., crashes when) obfuscating it. We have generated 500 instances of CarRace obfuscated with the same obfuscation, and we repeated the process for all the obfuscation, for a total of 500*44=22,000 distinct versions of CarRace.

To analyse the interaction of two or more factors we use two-way Analysis of Variance (ANOVA). We chose to use ANOVA because, differently from its non-parametric alternatives (such as the Kruskal-Wallis test) ANOVA allows to test for the presence of interactions between factors, i.e. it allows to perform a two-way analysis. Although ANOVA is a parametric test, it is considered quite robust also for non-normal and non-interval scale variables.

To study if the variation in the source code is due to non-determinism, we have applied the two-way Analysis of Variance (ANOVA) 10 times, once per metric.

For this analysis, only obfuscated code is considered, so clear code is not considered here.

Metric	Obfuscation	Iteration	Obfuscation:Iteration
McCane	<**0.01**	1.00	1.00
wmc	<**0.01**	0.98	1.00
dit	<**0.01**	0.99	1.00
noc	<**0.01**	0.98	1.00
cbo	<**0.01**	0.99	1.00
rfc	<**0.01**	1.00	1.00
lcom	<**0.01**	0.99	1.00
ca	<**0.01**	0.99	1.00
npm	<**0.01**	0.98	1.00
Lines of Code	<**0.01**	1.00	1.00

Table 12 Analysis of variance of Metric by Obfuscation and Iteration.

The results are summarized in Table 12, a distinct metric per row. The first row, for instance, reports the p-values of the Analysis of variance of McCabe complexity by Obfuscation and Iteration. For each metric (first column), the results of the test (p-values) should be interpreted as the source of the variation in the metrics (e.g., complexity) due to the fact that a different obfuscation is used (second column), due to the different iterations for each obfuscation (third column) or if variation in the metric is due an interaction (dependency) between the obfuscation and the iteration (last column).

The variation in the complexity due to the different obfuscation is statistically significant (in second column p-value <0.05), while the variation of complexity due to the 500 iterations with the same obfuscation is not significant (in the third column p-value is not <0.05). Eventually the obfuscation does not interact with the iteration to influence the complexity of the code (in the last column p-value is not <0.05). Exactly the same pattern occurs for the modularity and size metrics

All in all, while obfuscation influences complexity, modularity and size of code (see above in this section), we can not reject the null hypothesis H_{01nd} that non-determinism potentially involved by some obfuscating transformations does not influence the final result.

5.6 Analysis of Reinforcement

In the previous experiments, we considered combined obfuscation algorithms, applied with Allatori and Zelix Klassmaster tools. Allatory supports only two configurations of combined obfuscations (see Table 4) and individual obfuscations can not be applied separately. However, Zelix Klassmaster allows a more fine grained configuration, and single obfuscations can be applied separately.

Thus, in the following, we study the effect of individual obfuscations of Zelix Klassmaster, and we compare them with the two combined obfuscation configurations considered previously (i.e., *klass-grr* and *klass-light*). In detail, each combined configurations is compared with the single obfuscations that participate in the combination.

The effect of each obfuscation is assessed by comparing complexity, modularity and size of clear code and obfuscated code, using unpaired (two-tailed) Mann-Whitney test, with Holm correction (correction required because of multiple pairwise comparisons with overlapping data). While the statistical test allows for checking the presence of significant differences, it does not provide any information about the magnitude of such a difference. To this aim we adopt the notion of obfuscation *Potency*, as done previously in this section.

Table 13 reports the comparison for the more aggressive configuration (i.e., *klass-grr*), while Table 14 for the light configuration (i.e., *klass-light*). Obfuscations are reported in rows and metrics in columns. These tables report the effect of obfuscations as their *Potency*. However, to avoid misleading interpretations, Potency is reported only when the difference between clear and obfuscated code is statistically significant (according to the Mann-Whitney test with Holm correction). For example, the difference between code obfuscated with *klass-grr* and clear code measured as *ca* is not statistically significant, so no ca-potency is reported on the table for *klass-grr*. To help readability, for each metric, the highest value of potency is highlighted in boldface, while the lowest value is underlined.

Table 13 reports the comparison of the combined configuration *klass-grr* (first row) with the individual obfuscations composing it. Considering the code complexity (column *McCabe*) we see that potency of the compound obfuscation (0.93) is higher than most of the individual obfuscations, however the highest potency is scored by an individual obfuscation (1.15 for obfuscateFlow-aggressive).

This can be interpreted as an average reinforcement effect among individual obfuscations. Different strategies to make code more complex can be combined to make code even more complex. However, very elaborated strategies to achieve this objective can be combined at the cost of losing effectiveness, for example because specific classes could be targeted instead of all.

Considering modularity, we can observe reinforcement effect on coupling with respect to *wmc*, *cbo*, *rfc* and *npm*, where the potency of the combination is higher than the potency of all the individual obfuscations. Negative reinforcement is observed on cohesion, because the *lcom* potency of the combination is lower than the single obfuscations. For the remaining metrics, *noc* potency of the combination is comparable to

the value of individual obfuscations, while for *dit* and *ca* statistical significance is not observed.

The contrasting effect on coupling and cohesion could be interpreted as a deliberate strategy of the developers of Zelix Klassmaster to privilege the effect on coupling rather than on cohesion to make the code more hard to understand (apparently, this strategy can not be disabled by the configuration file).

Eventually, for the size we observe reinforcement, as the *LoC* potency of the combination (*klass-grr*) is much higher than the potency of single obfuscations.

Obfuscation	McCabe	wmc	dit	noc	cbo	rfc	lcom	ca	npm	LoC
klass-grr	0.93	**-0.19**		-0.11	**-0.06**	**-0.15**	-0.76		**-0.23**	**0.46**
aggressiveMethod-Renaming-true	0.64	-0.39	-0.02	-0.15	-0.22	-0.34	-0.61	-0.09	-0.44	0.15
keepInnerClassInfo-false	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15
keepGenericsInfo-false	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15
obfuscateFlow-aggressive	**1.15**	-0.39	-0.02	-0.15		-0.33	-0.62	-0.07	-0.44	0.30
encryptStringLiterals-aggressive	0.60	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.13
exceptionObfuscation-heavy	0.78	-0.39	-0.02	-0.15		-0.33	-0.61	-0.04	-0.44	0.20
autoReflection-Handling-normal	0.55	-0.36		**-0.10**	-0.29	-0.32	-0.18	**-0.18**	-0.40	0.21
lineNumbers-scramble	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.25
localVariables-delete	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15
randomize-true	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15
allClassesOpened-true	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15
deriveGroupingsFrom-InputChangeLog-false	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15

Table 13 Mann-Whitney test of complexity, modularity and size (with Holm correction) and potency for Zelix Klassmaster obfuscator (aggressive variant). For each metric, the highest value of potency is highlighted in boldface, while the lowest value is underlined.

Similarly to the *aggressive* case, Table 14 analyses the reinforcement effect of obfuscations for the light configuration of Zelix Klassmaster. Differently from the previous case, no relevant reinforcement is observed on complexity (*McCabe* potency) and size (*Loc* potency). In fact, the potency of the combination is quite low with respect to the potency of individual obfuscations. However, the trend on modularity is confirmed, as the reinforcement is positive on coupling (*wmc*, *cbo*, *rfc*, *ca*, *npm*) and negative on cohesion (*lcom*).

All in all, we can reject the null hypothesis H_{01r} and formulate these alternative hypotheses:

- When applying a combined obfuscation in Zelix Klassmaster, individual obfuscations reinforce each other to increase coupling potency (with respect to *wmc*, *cbo*, *rfc* and *npm*) and to decrease cohesion (with respect to *lcom*).
- When applying a combined obfuscation in Zelix Klassmaster, individual obfuscations reinforce each other to increase (*McCabe*) complexity potency and (*Loc*) size potency, but only when combining the more aggressive variant of obfuscations (i.e., klass-grr).

Obfuscation	McCabe	wmc	dit	noc	cbo	rfc	lcom	ca	npm	LoC
klass-light	0.31	**0.09**			**0.15**	**0.09**	-0.62	**0.14**	**0.01**	0.18
aggressiveMethod-Renaming-false	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15
keepInnerClassInfo-true	**0.77**	0.02			0.09	0.08	**-0.16**	0.09		**0.25**
keepGenericsInfo-true	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15
obfuscateFlow-light	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15
encryptStringLiterals-normal	0.28	-0.33	-0.02	-0.15	-0.22	-0.31	-0.58	-0.09	-0.44	0.03
exceptionObfuscation-light	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15
autoReflection-Handling-none	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15
lineNumbers-scramble	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	**0.25**
localVariables-delete	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15
randomize-false	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15
allClassesOpened-true	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15
deriveGroupingsFrom-InputChangeLog-false	0.64	-0.39	-0.02	-0.15	-0.22	-0.33	-0.61	-0.09	-0.44	0.15

Table 14 Mann-Whitney test of complexity, modularity and size (with Holm correction) and potency for Zelix Klassmaster obfuscator (light variant). For each metric, the highest value of potency is highlighted in boldface, while the lowest value is underlined.

6 RQ$_2$: Obfuscation Impact

In the previous section we were interested in a detailed analysis of the consequences of obfuscation on code quality. In this section we mean to analyze the impact of obfuscation on broader sense, but to achieve this objective we have to keep the level of the analysis less detailed.

For the second research question we formulate the subsequent null hypothesis:

H_{02i} : There is no difference in the *impact* of different obfuscations on the quality of source code.

Due to the large amount of data and p-values, it is not possible to show all detailed results dividing by obfuscation, by metric and by application, as has been done in Section 5 (however, detailed analyses are available in the technical report [3]).

Aggregated data are reported in Table 15, where obfuscations are displayed in rows and metrics are shown in columns. A cell reports the number of subject applications in which the Mann-Whitney test with Holm correction yields a statistically significant difference between clear code and code obfuscated with the obfuscation in the row, with respect to a metric in the column (non-zero values are reported in boldface to help readability). For example, the McCabe cyclomatic complexity (second column) of code obfuscated with *Allatori cfo* (first row) is statistically significantly different compared to the complexity of clear code in seven out of eighteen applications. Due to some implementation limits of the considered obfuscation tools, not all the obfuscations could be applied to all the applications. The number of applications where each obfuscation could be applied is shown in the "success" column. In this case, *Allatori cfo* could be applied on all the 18 subject applications, but other could not, for example *klass-grr* could be applied just on 13 applications.

The last column of the table summarizes the impact of the obfuscation as the number of metrics for which the obfuscation has shown a significant difference with respect to clear code. *Allatori cfo* reports an *impact* of 10 because obfuscated code was found to be significantly different than clear code in all the 10 considered metrics.

The obfuscations with higher impact are *Allatori-cfo*, *Allatory-lvn*, as they reported a significant difference with respect to clear code in all the 10 metrics. Another notable case is *Split Classes* (APP-sc) as it was significant with respect to 7 metrics.

Obfuscations *Klass-light*, *String Encoder* (APP-se) and *Static Method Bodies* (CL-smb) have been shown to be effective in causing significant differences in 5 metrics and *Klass-grr*, *Class Encrypter* (APP-ce) and *Array Folding* (APP-af) has an impact on 3 metrics.

All the remaining obfuscations have reported significant differences on 2 or less metrics, some of them did not significantly affect any metric.

All in all we can reject the null hypothesis H_{02i} and formulate the subsequent alternative hypothesis:

– Different obfuscations have different impact on the quality of source code. In particular, almost all the obfuscation transformations have an impact on (*LoC*) size and (McCabe) complexity and few obfuscations also have an impact on modularity as shown in Table 15.

7 RQ$_3$: Influence of Initial Quality

For the final research question concerning whether the initial quality of clear code influence the quality of obfuscated code, we formulate the subsequent null hypothesis:

H_{03q} : the potency of the obfuscation is not correlated with the quality of clear code.

7.1 Analysis of Correlation

To study the relation between the initial quality of the clear code the obfuscation potency, we use the *Pearson correlation* test. 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.

For each pair of metrics A and B, we compute the value of first metric in the clear code (i.e. $A(P)$) and the potency with respect to second metric (i.e., $T_{pot}(B, P)$). We repeated the same process for all the obfuscations T and all the programs P. Finally, for each obfuscation we apply the Pearson correlation test to study if there is a correlation between $A(P)$ and $T_{pot}(B, P)$ across all the case study programs, to study the influence on the initial value of metric A to the B potency due to obfuscation T.

For each obfuscation, we have tested the correlation between all the possible pairs metric-potency. Considering that the study involves 10 metrics, the complete pairwise correlation involve 100 pairs, so an appropriate correction factor is required. To this achieve this, we used the Holm correction factor.

Table 16 presents the results of the Pearson correlation with Holm correction, only statistically significant cases are reported. Three correlations are statistically significant for obfuscation *Klass-grr*, reported in the first three lines of the tables. For this

Obfuscation	McCabe	LoC	wmc	dit	noc	cbo	rfc	lcom	ca	npm	success	impact
allatori-cfo	**7**	**13**	**11**	**4**	**5**	**8**	**10**	**11**	**5**	**11**	18	10
allatori-lvn	**7**	**13**	**11**	**4**	**5**	**8**	**10**	**11**	**5**	**11**	18	10
klass-grr	**1**	**1**	0	0	0	0	0	0	**2**	0	13	3
klass-light	**1**	**1**	0	**1**	0	0	**1**	**1**	0	0	13	5
APP-bm	**1**	**8**	0	0	0	0	0	0	0	0	18	2
APP-ce	**2**	**1**	0	0	0	0	0	0	**1**	0	18	3
APP-fr	**1**	**8**	0	0	0	0	0	0	0	0	18	2
APP-rr	**1**	**8**	0	0	0	0	0	0	0	0	18	2
APP-af	**3**	**2**	0	0	**1**	0	0	0	0	0	4	3
APP-as	**1**	**1**	0	0	0	0	0	0	0	0	4	2
APP-BL	**2**	**2**	0	0	0	0	0	0	0	0	4	2
APP-cpr	**1**	0	0	0	0	0	0	0	0	0	6	1
APP-di	**3**	0	0	0	0	0	0	0	0	0	5	1
APP-ias	**1**	**2**	0	0	0	0	0	0	0	0	9	2
APP-im	0	**1**	0	0	0	0	**1**	0	0	0	6	2
APP-on	0	**2**	0	0	0	0	0	0	0	0	7	1
APP-pa	**1**	0	0	0	0	0	0	0	0	0	8	1
APP-sc	0	0	0	0	0	0	0	0	0	0	6	0
APP-se	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	4	5
CL-cs	**2**	**1**	**1**	0	**3**	0	**1**	0	**1**	1	6	7
CL-ob	0	**4**	0	0	0	0	0	0	0	0	8	1
CL-sop	**6**	**7**	0	0	0	0	0	0	0	0	13	2
CL-fa	**1**	**8**	0	0	0	0	0	0	0	0	18	2
CL-mm	**1**	**8**	0	0	0	0	0	0	0	0	18	2
CL-pf	**1**	**8**	0	0	0	0	0	0	0	0	18	2
CL-smb	**18**	**17**	**13**	0	0	0	**5**	**17**	0	0	18	5
MET-bs	**1**	0	0	0	0	0	0	0	0	0	8	1
MET-bsp	**1**	0	0	0	0	0	0	0	0	0	6	1
MET-il	0	0	0	0	0	0	0	0	0	0	7	0
MET-iop	**1**	0	0	0	0	0	0	0	0	0	6	1
MET-ir	0	0	0	0	0	0	0	0	0	0	6	0
MET-ppt	**1**	0	0	0	0	0	0	0	0	0	8	1
MET-ri	**1**	0	0	0	0	0	0	0	0	0	6	1
MET-rp	**1**	**4**	0	0	0	0	0	0	0	0	8	2
MET-vr	**1**	0	0	0	0	0	0	0	0	0	12	1
MET-bc	**1**	**1**	0	0	0	0	0	0	0	0	4	2
MET-bi	**5**	**11**	0	0	0	0	0	0	0	0	18	2
MET-dr	**1**	**8**	0	0	0	0	0	0	0	0	17	2
MET-mli	**1**	**8**	0	0	0	0	0	0	0	0	17	2
MET-obi	**3**	**15**	0	0	0	0	0	0	0	0	17	2
MET-ppr	**1**	**12**	0	0	0	0	0	0	0	0	17	2
MET-rdc	**1**	**14**	0	0	0	0	0	0	0	0	17	2
MET-tbi	**7**	**8**	0	0	0	0	0	0	0	0	17	2

Table 15 Mann-Whitney test (with Holm correction) between clear and obfuscated code, and impact of obfuscation.

obfuscation noc, McCabe complexity and number of Lines of Code are highly correlated with the potency computed as the increase of McCabe complexity of the obfuscated code. This means that *Klass-grr* is a potent obfuscation because it makes the complexity of code increase, but the increment in complexity depends on the initial quality of clear code. Added complexity is higher when the classes in the clear code have high coupling (noc), high (McCabe) complexity and large size (LoC).

Obfuscation *Dynamic Inliner* (APP-di) presents three cases with significant correlation with the cbo potency (i.e. potency computed on cbo). However, while correlation

Obfuscation	Metric	Potency	Correlation
klass-grr	noc	Pot_McCabe	0.67
klass-grr	McCabe	Pot_McCabe	0.82
klass-grr	LoC	Pot_McCabe	0.86
APP-di	wmc	Pot_cbo	-1.00
APP-di	dit	Pot_cbo	0.99
APP-di	npm	Pot_cbo	-1.00
APP-im	noc	Pot_cbo	0.95
APP-im	noc	Pot_ca	0.95
APP-on	npm	Pot_cbo	0.97
APP-on	npm	Pot_ca	0.98
APP-sc	noc	Pot_dit	0.98
APP-sc	lcom	Pot_lcom	1.00
APP-se	LoC	Pot_wmc	-0.97
CL-smb	npm	Pot_rfc	0.73
MET-bs	noc	Pot_ca	-0.99

Table 16 Pearson correlation test between initial metric and potency (with Holm correction).

between dit and cbo potency is positive, the correlation between wmc/npm and cbo potency is negative. This means that this obfuscation scores higher cbo potency on those applications whose classes present high values of dit, but low values of wmc/npm.

Other obfuscations whose potency is correlated with modularity and size of classes in clear code are *Interleave Methods* (APP-im), *Overload Names* (APP-on), *Split Classes* (APP-sc), *String Encoder* (APP-se), *Static Method Bodies* (CL-smb) and *Bludgeon Signatures* (MET-bs).

Overall, many cases of significant correlation between the initial quality of clear code and the potency of obfuscating transformations can be observed. Thus, we can reject the null hypothesis H_{03q} and formulate the following alternative hypothesis:

– The potency of obfuscating transformations is correlated with the quality of clear code for the cases shown in Table 16.

8 Discussion

Here we report our main findings and we elaborate on some implications for developers who want to use code obfuscation and who need to decide which particular algorithm to adopt.

8.1 Findings

The main findings of this research are as follows:

– *Opposite obfuscation strategies for complexity*: even if the intended purpose of source code obfuscation should be to make code more complex to understand and attack, this objective does not map straightforward to one specific code complexity metric (such as McCabe cyclomatic complexity). As a matter of fact, different obfuscating transformations take opposite strategies with code complexity to make it hard to understand. Some obfuscations increase the cyclomatic complexity of code, while other ones decrease it, probably with the final purpose of obstructing understanding

in other ways (see RQ_1, analysis of complexity). For example, Klass-grr increase the cyclomatic complexity of obfuscated code, while *Class Splitter* make the cyclomatic complexity decrease.

- *Cohesion Vs coupling*: the results here are similar to what has been observed for code complexity, also when considering modularity an opposite effect is expected on cohesion and coupling. In fact, to make the code harder to understand, intuitively cohesion should be low and coupling should be high. However, we observed that many obfuscations were able to decrease cohesion (lcom increased), but just few transformations had a high coupling potency (see RQ_1, analysis of modularity). Moreover, some cases worked well on different directions on different coupling metrics. For example, *Class Splitter* has high number of children (noc) coupling potency, but low coupling potency on all the other coupling metrics.
- *Contrasting effect of code size*: Code size metrics measure the amount of code that an attacker should consider to perpetrate an attack. We observed that obfuscation might either increase or decrease the size of obfuscated classes to make code harder to understand (see RQ_1, analysis of size). Probably, the effect of size reduction is just a side effect of a transformation that aims at working on other aspects relevant to code understanding. This is the case of *Class Encrypter*, whose objective is to encrypt classes and remove useful information, thus achieving also size reduction. However, size reduction in this case is not expected to bring major benefits to understanding.
- *Obfuscation could reinforce each other*: when combining different obfuscations to protect the code, the effect is not a straightforward reinforcement of each other transformation. In fact, in Zelix Klassmater we observed that combined obfuscations helps in increasing coupling, at the cost of higher cohesion (lower *lcom*). Instead, the effect on complexity and size really depends on the composition configuration. As a matter of fact, we observed contrasting reinforcement effects on complexity and size on different composition configurations (See RQ_1, analysis of reinforcement).
- *Impact of commercial tools depends on the configuration*: tools can be complex to use and developers should be quite familiar with them to obtain satisfying results. In fact, even if commercial tools guarantee high impacts, their performance highly depends on configuration parameters. As a matter of fact, the impact of Zelix Klassmaster almost doubles, increasing from 3 to 6 when moving from a light to a more aggressive configuration (see RQ_2, analysis of impact).
- *Initial quality of clear code affects obfuscation potency*: Different developers adopt different programming styles, so distinct programs vary considerably in terms of code metrics (see QR_2, analysis of diversification among subject applications). Additionally, the capability of obfuscation in altering code is not an absolute measure, but it may depend on the properties of the code to obfuscate. For instance, the presence of a particular feature in the clear code may influence the potency of an obfuscation that relies on such a feature. In particular, if the clear code contains few string variables, then the effect of the *String Encoder* obfuscation is expected to be relatively marginal, when compared to the effect of the same obfuscation on another program that process many strings. Seemingly, if the code does not contain generics or arrays, the obfuscations relying on these features will not be very effective.

  Moreover, if the original code is already obscure, because of high complexity, tangled modularity and large size classes, and the obfuscation is able increase even

more the complexity, then the final complexity can be huge, even if the measured potency (difference between clear and obfuscated) might not be so high. Conversely, if the original code is not so complex and the obfuscation is able to add complexity, the measured potency could be higher than the previous case, even if the absolute complexity of the result is lower than the previous case.
In particular, the measurable effect of code obfuscation depends on the initial size and complexity of the clear code (see QR_3, analysis of correlation). Thus, to maximize obscurity, project managers should carefully evaluate which obfuscation to adopt, by considering what are the programming features currently adopted by the developer team.

8.2 Recommendations

Based on these results, we can formulate some implications for developers who have to decide which obfuscation transformation to adopt. First of all, a developer should carefully elicit the security requirements of the application to be protected. This involves thinking of the assets to protect (e.g., secret keys or critical functionalities) and modelling the attack's behavior.

In the case where the attack requires substantial code understanding, code obfuscation could be an effective way to delay such a human-intensive process. But the obfuscation to deploy should be selected carefully, depending on the qualities of the code that are important to change. In this study we observed the effect of obfuscation on complexity, modularity and size, so our results support the choice of obfuscation with respect to these dimensions.

8.2.1 Complexity Potency

When a developer is interested in delivering obfuscated code that is very complex (with respect to cyclomatic complexity), obfuscations with high complexity potency should be selected. Luckily, many of the considered obfuscating transformations have positive potency, they increase the McCabe cyclomatic complexity of the obfuscated code. In particular, three obfuscations report a potency >0.33, they are *Klass-grr* with a potency of 0.93, *Interleave Methods* (APP-im) with 0.68 and *Simple Opaque Predicates* (CL-sop) with 0.42. So these three obfuscations should be selected to make obfuscated code more complex.

However, a developer should keep in mind that the complexity potency of *Klass-grr* is very sensitive to the initial quality of clear code. To produce very complex obfuscated code when using this transformation, the clear code should have high (noc) coupling, a large (Loc) size and already be quite complex, i.e. with a high value of McCabe cyclomatic complexity. Moreover, *Klass-grr* has also relevant effects on size (size potency=0.46) but not on modularity, because it makes coupling decrease and modularity increase.

It is interesting to note that the reduced effectiveness of *Klass-light*, i.e. the less aggressive variant of *Kalss-grr*, can be quantified with a reduction of 66% in complexity potency, from 0.93 to 0.31.

Similarly, also *Interleave Methods* (APP-im) is effective on size but not on modularity.

Conversely, *Simple Opaque Predicates* (CL-sop) is effective on size and on modularity, but limited to coupling (cohesion potency is still a problem).

8.2.2 Modularity Potency

In case the code needs to be protected by resorting on degraded modularity, those obfuscation with positive coupling potency and positive cohesion potency (measure on lcom) should be selected.

Some obfuscations present a relevant coupling potency, because they make obfuscated classes much more coupled than in the clear code, with the aim of obstructing code comprehension. However, coupling is measured by multiple metrics. If we consider wmc, the most relevant positive case is *Static Methods Bodies* (CL-smb) with a potency of 0.78. This obfuscation also has good values for cohesion (potency=2.78) but the drawback is in complexity and size where the potency is negative. Thus, *Static Method Bodies* should be used when modularity is the main objective. Finally, the coupling potency (with respect to rcf) of this obfuscation is highly correlated with the npm of the clear code.

A relevant case of dit potency for *Class Encrypter* (APP-ce, potency=1.02) has been observed, but the potency is negative when considering other coupling metrics, such as cbo and ca coupling. Additionally size potency for *Class Encrypter* is negative.

Moving to noc coupling (i.e., Number of Children), *Class Splitter* (CL-cs) shows high potency (1.09), but the potency for all the other metrics is negative (including modularity, complexity and size). So, this obfuscation is recommended where the developer objective is to complicate the inheritance relation among classes.

For cbo, rfc, ca, and npm, the potency with higher absolute values are all negatives. For cbo coupling, they are *Class Encoder* (APP-ce) with a potency of -1 and *Array Folding* (APP-af), *Array Splitting* (APP-as) and *Bloat* (APP-bl) with a potency of -0.73. Whereas, for rfc coupling highest absolute potency values are observed on *Allatori-cfo*, *Allatori-lvm*, *Interleave Methods* (APP-im) and *Class Splitter* (CL-cs) with values respectively of -0.35, -0.35, -0.37 and -0.41. Ca coupling has a high negative potency on *Class Encrypter* (APP-ce, ca potency=-1.00), and npm coupling has a high negative potency on *Allatori-cfo* and *Allatori-lvm*, where the values reached -0.42.

Considering (lcom) cohesion, we have to observe that lcom has an inverse meaning. In fact, *Lack of Cohesion in Methods* is high when cohesion is low, so a high value of this metric is expected to correspond to code to that is more difficult to understand. Some obfuscations show a very high cohesion potency, *Overload Names* (APP-on) has a lcom potency of 9.43, while *Objectify* (CL-ob), *Bludgeon Signatures* (MET-bs) *Promotion Primitive Registers* (MET-ppt) and *Reorder Parameters* (MET-rp) reported a lcom potency between 3 and 4. For *Parameter Alias* (APP-pa) and *Static method Bodies* (CL-smb) the lcom potency is between 2 and 3.

Among these, *Overload Names* (APP-on) is the only one that shows poor potency on all the other considered metrics, so this obfuscation should be used when cohesion is the only protection goal.

Among the other obfuscation with high lcom potency, *Objectify* (CL-ob), *Parameter Alias* (APP-pa) and *Promotion Primitive Registers* (MET-ppt) have positive but small potency on size, wmc and npm coupling, while *Reorder Parameters* (MET-rp) and *Bludgeon Signatures* (MET-bs) have positive potency only on wmc and npm. *Static method Bodies* (CL-smb) shows high potency on rfc wmc.

Only two obfuscations are able to significantly affect all the modularity metrics, they are *Integer Array Splitting* (App-ias) and *Class Splitter* (CL-cs). However, different coupling metrics are affected in different directions, some increases and other decreases.

8.2.3 Size Potency

Only a few obfuscations are not effective with respect to the change of size of classes from clear code to obfuscated code. A large increase in the size (positive potency) is scored by *Interleave Methods* (APP-im, potency=0.78) and *Buggy Code* (MET-bc, potency=0.61), because these obfuscation add code to make program understanding harder, so they should be adopted when increased code size is among the objectives of the developer. *Interleave Methods* also has complexity potency, but negative potency when considering modularity. To some extent, *Buggy Code* represents a better choice, because its complexity potency is high as well, but additionally it shows reasonable values of rfc and cbo potency.

Conversely, a significant size reduction is observed when obfuscating code with *Class Encrypter* (CL-ce), *Bloat* (APP-bl), *String Encoder* (APP-se) and *Array Folding* (APP-af) respectively with potency of -0.61, -0.60, -0.60 and -0.59. So these obfuscations should not be used when aiming for size increases. However, all these obfuscations involve size reduction because they aim at removing relevant information from the code, information that an attacker may use in an attempt to reverse engineer the program.

Size potency is never significantly influenced by the size, complexity nor modularity of the clear code. In fact, extra code can always be added to increase classes size. Moreover, the possibility of optimizing code and reducing size may depend on code qualities not captured by modularity and complexity metrics, and result in reduced class sizes.

8.3 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 many code metrics, devoted to measuring different aspects of the quality of code (complexity, modularity and size). Moreover, the conclusions have been drawn based on objective statistical tests. When possible, we have adopted non-parametric tests (such as Mann-Whitney) that do no make assumptions on the normal distribution of data. Furthermore, we have used the ANOVA test which, although parametric, is considered robust against deviations from normality. In all the cases where multiple pairwise comparisons are performed with overlapping data, we control the increase probability of committing type I error (i.e. rejecting a true null hypothesis) by adopting the Holm correction.

Moreover, while obfuscators works on compiled bytecode, to compute source-level metrics we have had to decompile obfuscated code. This process might alter the struc-

ture of obfuscated bytecode, however this is a process that an attacker would probably also need to undertake.

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 44 different obfuscation algorithms and source code from different domains, counting up for more than 4 million lines of code. Moreover, applications have been selected from different repositories, including also the 10 of the most popular applications from Source-Forge. Nonetheless, only further experiment with other obfuscator tools[13] and more subject applications can confirm our findings.

9 Conclusion and Further Work

Code obfuscation has measurable and visible effects. This paper presents the results of a large scale study to quantify the effects of various obfuscation techniques, as measured by several metrics on Java code. We have considered 44 algorithms for source code obfuscation. 40 of them are implementation from Sandmark, an open source tool. Furthermore 4 implementations are from commercial tools: 2 configurations are produced by the Allatori toolset and 2 configurations by Zelix Klassmaster. The consequences of code obfuscation have been measured using 10 different metrics, considering the modularity, the size and the complexity dimension. The study has involved more that 4 millions lines of code of Java.

Our findings show that code obfuscation impacts all the considered metrics, however different algorithms have different effects. Moreover, the initial quality of clear code to be obfuscated influences the results of obfuscation. The present paper is meant to shed some light on what are the features of the available obfuscating transformations. Project managers and developers should evaluate them carefully, when they have to decide which algorithm to adopt to obfuscate Java code.

As future work, we intend to study combinations of obfuscations also from different tools, and compare the effect of the combination with the single effect of the composing obfuscation tools. Moreover, we plan to define new metrics more related to the attacker behavior; for example as method calls to java libraries (like GUI, network) cannot be obfuscated, and these are a good indicator of how many starting points are available to the attacker to analyze code, thus a metric can be defined to measure such "weakness" in the original code and how the obfuscation can reduce it. On the other hand collusion attacks are also an important strand to be researched: subsequent versions of a program are often used to find differences in code versions (to detect the location of vulnerabilities in the old version, patched in the new version), or detecting similarities between versions that can be used to spot relevant code to be used as starting point for analysis. Diversity of software obfuscations has become important to mitigate such attacks, but metrics must be identified or created to evaluate the effectiveness of obfuscation throughout time by measuring the effective diversity between versions.

Moreover, we plan to involve programmers in controlled experiments, to validate the extent that the metrics considered in this study are correlated to the effort required to understand and attack Java code.

[13] Available obfuscation tools are ProGuard, yGuard, JODE, JavaGuard, RetroGuard, jarg, etc

Acknowledgements

The authors would like to thank Marco Torchiano for the interesting discussion on the analysis procedure and the Zelix Klassmaster™ developers for the full evaluation copy of their tool and the feedback provided.

References

1. Anckaert, B., Madou, M., De Sutter, B., De Bus, B., De Bosschere, K., Preneel, B.: Program obfuscation: a quantitative approach. In: Proceedings of the 2007 ACM workshop on Quality of protection, QoP '07, pp. 15–20. ACM, New York, NY, USA (2007). DOI 10.1145/1314257.1314263. URL http://dx.doi.org/10.1145/1314257.1314263
2. Basili, V., Briand, L., Melo, W.: A validation of object-oriented design metrics as quality indicators. Software Engineering, IEEE Transactions on **22**(10), 751–761 (1996)
3. Ceccato, M., Capiluppi, A., Falcarin, P., Boldyreff, C.: A large study on the effect of code obfuscation on the quality of java code: Detailed analysis of data. Tech. rep., FBK, TR-FBK-SE-2013-3, http://se.fbk.eu/en/techreps (2013). URL http://se.fbk.eu/sites/se.fbk.eu/files/TR-FBK-SE-2013-3.pdf
4. Ceccato, M., Di Penta, M., Nagra, J., Falcarin, P., Ricca, F., Torchiano, M., Tonella, P.: Towards experimental evaluation of code obfuscation techniques. In: Proceedings of the 4th ACM workshop on Quality of protection, QoP '08, pp. 39–46. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1456362.1456371. URL http://doi.acm.org/10.1145/1456362.1456371
5. Ceccato, M., Penta, M., Falcarin, P., Ricca, F., Torchiano, M., Tonella, P.: A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. Empirical Software Engineering pp. 1–35 (2013). DOI 10.1007/s10664-013-9248-x. URL http://dx.doi.org/10.1007/s10664-013-9248-x
6. Ceccato, M., Penta, M.D., Nagra, J., Falcarin, P., Ricca, F., Torchiano, M., Tonella, P.: The effectiveness of source code obfuscation: An experimental assessment. In: ICPC, pp. 178–187. IEEE Computer Society (2009)
7. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**, 476–493 (1994). DOI 10.1109/32.295895. URL http://dl.acm.org/citation.cfm?id=630808.631131
8. Cohen, F.B.: Operating system protection through program evolution. Comput. Secur. **12**, 565–584 (1993). DOI 10.1016/0167-4048(93)90054-9. URL http://dl.acm.org/citation.cfm?id=179007.179012
9. Collberg, C., Myles, G., Huntwork, A.: Sandmark–a tool for software protection research. IEEE Security and Privacy **1**, 40–49 (2003). DOI 10.1109/MSECP.2003.1219058. URL http://dl.acm.org/citation.cfm?id=939830.939941
10. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Tech. Rep. 148 (1997). URL http://www.cs.auckland.ac.nz/collberg/Research/Publications/CollbergThomborsonLow97a/index.html. Http://www.cs.auckland.ac.nz/∼collberg/Research/Publications/CollbergThomborsonLow97a/index.html
11. Collberg, C.S., Thomborson, C.: Watermarking, tamper-proofing, and obfuscation: tools for software protection. IEEE Trans. Softw. Eng. **28**, 735–746 (2002). DOI 10.1109/TSE.2002.1027797. URL http://dl.acm.org/citation.cfm?id=636196.636198
12. Falcarin, P., Collberg, C., Atallah, M., Jakubowski, M.: Guest editors' introduction: Software protection. IEEE Softw. **28**, 24–27 (2011). DOI http://dx.doi.org/10.1109/MS.2011.34. URL http://dx.doi.org/10.1109/MS.2011.34
13. Goto, H., Mambo, M., Matsumura, K., Shizuya, H.: An approach to the objective and quantitative evaluation of tamper-resistant software. In: Proceedings of the Third International Workshop on Information Security, ISW '00, pp. 82–96. Springer-Verlag, London, UK (2000). URL http://dl.acm.org/citation.cfm?id=648024.744206
14. Heffner, K., Collberg, C.: The obfuscation executive. In: Information Security, pp. 428–440. Springer (2004)
15. Hosking, A.L., Nystrom, N., Whitlock, D., Cutts, Q., Diwan, A.: Partial redundancy elimination for access path expressions. Software: Practice and Experience **31**(6), 577–600 (2001). DOI 10.1002/spe.371. URL http://dx.doi.org/10.1002/spe.371

16. Jakubowski, M.H., Saw, C.W., Venkatesan, R.: Iterated transformations and quantitative metrics for software protection. In: SECRYPT, pp. 359–368 (2009)
17. Jureczko, M., Spinellis, D.: Using Object-Oriented Design Metrics to Predict Software Defects, *Monographs of System Dependability*, vol. Models and Methodology of System Dependability, pp. 69–81. Oficyna Wydawnicza Politechniki Wroclawskiej, Wroclaw, Poland (2010)
18. Karnick, M., MacBride, J., McGinnis, S., Tang, Y., Ramachandran, R.: A qualitative analysis of java obfuscation. In: Proceedings of 10th IASTED International Conference on Software Engineering and Applications, Dallas TX, USA (2006)
19. Kouznetsov, P.: Jad - the fast JAva Decompiler. URL `http://www.kpdus.com/jad.html`
20. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM conference on Computer and communications security, CCS '03, pp. 290–299. ACM, New York, NY, USA (2003). DOI http://doi.acm.org/10.1145/948109.948149. URL `http://doi.acm.org/10.1145/948109.948149`
21. Lv, Z., Ri, S., Uhvhdufk, D.E., Dw, D., Wkh, Y., Ri, X., Srsxodu, W., Zrun, Q.D.S., Vkrzhg, Z.H.: On the relationship between cyclomatic complexity and oo ness. 9th ECOOP Workshop on Quantitative Approaches in ObjectOriented Software Engineering (2005)
22. McCabe, T.J.: A complexity measure. IEEE Trans. Software Eng. pp. 308–320 (1976)
23. Sheskin, D.: Handbook of Parametric and Nonparametric Statistical Procedures (4th Ed.). Chapman & All (2007)
24. Simon, F., Steinbrückner, F., Lewerentz, C.: Metrics based refactoring. In: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, CSMR '01, pp. 30–. IEEE Computer Society, Washington, DC, USA (2001). URL `http://dl.acm.org/citation.cfm?id=794203.795287`
25. Sutherland, I., Kalb, G.E., Blyth, A., Mulley, G.: An empirical examination of the reverse engineering process for binary files. Computers & Security **25**(3), 221–228 (2006)
26. Udupa, S.K., Debray, S.K., Madou, M.: Deobfuscation: Reverse engineering obfuscated code. In: Proceedings of the 12th Working Conference on Reverse Engineering, pp. 45–54. IEEE Computer Society, Washington, DC, USA (2005). DOI 10.1109/WCRE.2005.13. URL `http://dl.acm.org/citation.cfm?id=1107841.1108171`
27. Vasa, R., Schneider, J.g.: Evolution of cyclomatic complexity in object oriented software. Proceedings of 7th ECOOP Workshop on Quantitative Approaches in ObjectOriented Software Engineering QAOOSE 03 pp. 1–5 (2003). URL `http://www.it.swin.edu.au/personal/jschneider/Pub/qaoose03.pdf`
28. Visaggio, C.A., Pagin, G.A., Canfora, G.: An empirical study of metric-based methods to detect obfuscated code
29. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A.: Experimentation in Software Engineering - An Introduction. Kluwer Academic Publishers (2000)
30. Wyseur, B.: White-box cryptography. Ph.D. thesis, Katholieke Universiteit Leuven (2009). URL `http://www.cosic.esat.kuleuven.be/publications/talk-98.pdf`
31. Zeng, Y., Liu, F., Luo, X., Yang, C.: Software watermarking through obfuscated interpretation: Implementation nad analysis. Journal of Multimedia **6**(4), 329–339 (2011)