From Obfuscation to Comprehension

Eran Avidan Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University of Jerusalem, 91904 Jerusalem, Israel

***Abstract*—Code obfuscation techniques are widely used in industry to increase protection of source code and intellectual property. The idea is that even if attackers gain hold of source code, it will be hard for them to understand what it does and how. Thus obfuscation techniques are specifically targeted at human comprehension of code. We suggest that the ideas and experience embedded in obfuscations can be used to learn about comprehension. In particular, we survey known obfuscation techniques and use them in an attempt to derive metrics for code (in)comprehensibility. This leads to emphasis on issues such as identifier naming, which are typically left on the sidelines in discussions of code comprehension, and motivates increased efforts to measure their effect.**

I. Introduction

Software is often complicated and hard to understand. Brooks divides the complexity of software into two: "essential" complexity which arises from the system's requirements, and "accidental" complexity related to how it is expressed [1]. The essential complexity reflects the hardness of the problem, and cannot be abstracted away, as in complex legal requirements in an accounting application. But accidental complexity can and should be reduced, e.g. by using design patterns, refactoring, readability-enhancing transformations, and documentation. These improve the representation and make the code more comprehensible and maintainable. Code complexity metrics are also related to this, being used to identify code segments which are overly complicated and may benefit from additional work.

Obfuscation is the opposite activity: the purposeful degradation of the code in order to prevent reverse engineering, and thereby protect the intellectual effort invested in creating the program in the first place. Over the years a large body of knowledge on code obfuscation has been accumulated, including the development of new obfuscation techniques, their classification and description, and empirical work on their effectiveness. Several code obfuscators for different languages are available commercially or as open-source products.

Naturally, there are many connections between obfuscation, code complexity, and program comprehension. For example, code complexity metrics have been used to evaluate the potency of obfuscation techniques, based on the premise that more complex code is harder to decipher. But there has been little interaction. Obfuscation has its own community, publications, empirical studies, and even events such as *The International Obfuscated C Code Contest* which has been held annually since 1984. Thus it seems reasonable that comprehension can learn from obfuscation, and that ideas developed and evaluated for obfuscation research can be used to advance our understanding of program comprehension.

To explore this possible connection, we start with a short primer on obfuscation in Section II. Section III discusses elements of comprehension and outlines ideas about how comprehension can learn from obfuscation. Section IV then lists specific insights about comprehension that are derived from obfuscation, and suggests various code attributes that should be studied and quantified.

II. Code Obfuscation

Reverse engineering uses program analysis techniques, such as discussed in Nielson et al. [14], in the process of extracting knowledge and information embedded in the program. One way of making a program more resilient to these techniques is applying code obfuscation transformations to the code.

Collberg et al. [7] were the first to formally define obfuscation. Their definition was that obfuscation is a semantic-preserving transformation function O which maps a program P to $O(P)$ such that $O(P)$ must have the same observable behavior. In practice, obfuscation transformations are applied in order to make P harder for program analysis. They classified obfuscation transformations into four main classes: layout, data, control-flow, and preventive.

Layout transformations change or remove useful information from the source code without changing its behavior. Specific types of layout obfuscations include identifiers renaming, changing of formatting, and comments removal [7]. *Data* transformations target data structures contained in the program, and affect the storage, encoding, aggregation, or ordering of the data. Specific examples are converting static data to procedures, change of encoding, and array merging [7]. *Control-flow* obfuscations attempt to obscure the control flow of the program. These are classified as affecting aggregation, ordering, or computation of the flow of control, and many of them rely on opaque variables or predicates. Examples of control-flow obfuscations are dead code generation [3], method merging [3], and class coalescing [17]. *Preventive* obfuscations are transformations that are specifically targeted at deobfuscation or static analysis techniques, with the goal of making them more difficult to perform.

In order to evaluate the quality of an obfuscation transformation Collberg et al. suggested the potency, resilience, and cost metrics [7]. Later, Low added the stealth measure [12].

Potency quantifies the degree to which an obfuscation makes it harder for a human to understand the code. In other words, it reflects how much $O(P)$ is more obscure than

P. Though this ideally relies on human cognitive abilities, some known software complexity metrics (such as Halstead's program metrics or McCabe's cyclomatic complexity) can also be applied to $O(P)$ to show that it is more complex than P. *Resilience* measures how well a transformation holds up against an automatic deobfuscator. For example, most formatting transformations can be trivially undone by a source code beautifier, so they are not resilient. *Execution cost* is the execution time and space penalty of $O(P)$ compared to P, and *Stealth* is the degree to which the code resulting from an obfuscating transformation differs from the original code.

III. Elements of Comprehension

Programming languages are different from natural languages in that a syntactically legal piece of code can have only one meaning: the meaning understood by the compiler, and reflected in the code's execution. Nevertheless, it may still be hard for a human to comprehend the code and understand what it does at the conceptual level, especially if the code is complex or obfuscated.

Rather than trying to formally define comprehension, we suggest a list of elements that may influence comprehension. These will be used later in connection to obfuscations. The elements we identify are readability, design, flow complexity, and transparency.

Readability is a local property of the code. Readable code has reasonably short lines, consistent indentation, and meaningful names of variables and functions. Buse and Weimer [2] have shown how to produce a metric for software readability based on judgments of human annotators. This metric exhibits significant correlation with more conventional metrics of software quality, such as defect reports and code churn. More abstract models of readability have also been proposed [15].

In *design* we refer mainly to modularity, namely the decision how to structure the code and assign responsibilities to different modules. This reflects issues such as information hiding, module cohesion, and reduced coupling between independent modules. Modular code is expected to be easier to comprehend because modules can be studied in isolation.

Flow complexity reflects the dynamic side of the design — how the modules communicate with each other at run-time. A prime example is fan-in and fan-out, namely the pattern that modules call each other [10]. It also includes local code complexity metrics such as McCabe's cyclomatic complexity (MCC) [13] or the use of gotos [9].

Transparency refers to the degree that the code directly reflects the algorithmic ideas it is supposed to implement. In transparent code, the mapping from concepts to code is direct and self-evident, and as a result the code is easier to comprehend. In contrast, opaque code is code where the underlying concepts are hidden behind myriad details making the code harder to comprehend. For example, in advocating for structured programming Dijstra writes [9] "we should do [...] our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible."

Obfuscations make code harder to understand by impairing these elements. We argue that the study of code comprehension can benefit from insights derived in the context of code obfuscation. For example, Ceccato et al. [4] showed that comprehension tasks on code obfuscated with identifier renaming require, on average, twice more time than on code obfuscated with opaque predicates. This suggests that appropriate identifier names have a higher impact on comprehension than maintaining a simple control flow. Consequently it motivates new directions in complexity metrics: instead of focusing on easily measured static code properties, it is necessary to also measure how well variable names reflect the code's semantics. In the next section we suggest initial ideas along this line.

IV. Obfuscation-Based Insights About Comprehension

We will now list a few insights regarding comprehension inspired by specific obfuscation transformations. Despite the origin from explicit obfuscations, each one of them could surly occur also as a result of bad practice, a developer error, or just the cumulative uncoordinated work of multiple people during evolution. The insights are expressed as measurable code attributes, so they readily translate into complexity metrics, where higher values indicate harder to comprehend code. These metrics can usually be normalized to the range [0,1] by dividing by their maximal possible value. They can be measured in various contexts, e.g. a class, method, or any other level of hierarchy in the program.

Name reuse [5]. [Impair: Transparency]

The idea of this obfuscation is to reuse the same identifier as many times as possible, so that one cannot know what it refers to just from the name. We will focus on reuse in a specific class where an identifier can denote several local variables, fields, and methods at the same time. As a result, when one tries to understand what the variable means, it is necessary to take the context into consideration as well as the way the identifier is used.

Definition: Let C be some class in the program, and ID the set with all identifiers in C. ID can be partitioned into the set of all local variable names L, the set of all class field names F, and the set with all method names M, all in reference to C. We shall define two different sets of pairs as follows. (1) LF: Local variables and fields with the same name, also known as shadowing fields with variables. (2) MM: Methods with the same name, also known as method overloading. More formally: $LF = \{(l_i, f_j) \mid l_i \in L, f_j \in F, l_i = f_j\}$, $MM = \{(m_i, m_j) \mid m_i, m_j \in M, m_i = m_j, i < j\}$.

LF is known to be confusing and discouraged by some code quality management platforms, such as SonarQube [16]. Even so, there is no known metric that measure the level of LF in a class. We suggest the Shadowed Field Metric:

$$SFM = |LF|$$

Scale: $0 \leq SFM \leq |F|$, where $|F|$ means that all the fields are shadowed by local variables in at least one class method, and 0 that none of them are.

Another metric that could reflect on possible confusion would count all the names reuse except method overloading.

Definition: Let P the set of all possible identifier pairs and EQ the set of all the pairs with equal names. More formally: $P = \{(id_i, id_j) \mid id_i, id_j \in ID, i < j, (id_i, id_j) \notin MM\}$, $EQ = \{(id_i, id_j) \mid (id_i, id_j) \in P, id_i = id_j\}$. Then, our Name Reuse Metric will be:

$$NRM = |EQ|$$

Scale: $0 \leq NRM \leq |P|$, where $|P|$ is when all identifiers are the same, and 0 is when every element has its unique identifier.

Using similar names. [Impair: Readability]

A variant of the above is to use very similar names, e.g. long meaningless strings that differ in a single character, such as iiiiiij and iiiiiiij. This may create confusion, making it hard to tell apart different variables, methods, or any other pairs of elements in the program. In order to evaluate the similarity of two strings we use the edit distance: the minimum number of insertions, deletions, and substitutions required to transform one string into the other.

Definition: Let ID be a set of all identifiers, and ed_{ij} the edit distance between id_i and id_j where $id_i, id_j \in ID$. We define sim_{ij} as the measure of similarity between id_i and id_j:

$$sim_{ij} = \frac{max(|id_i|, |id_j|) - ed_{ij}}{max(|id_i|, |id_j|)}$$

Given this, the metric will be the number of identifier pairs which have a high similarity:

$$SIMM = |\{(i,j) \mid i < j, sim_{ij} > a\}|$$

Scale: $0 \leq SIMM \leq (n^2 - n)/2$, where 0 is when no pair of names have a similarity larger than a, and $(n^2 - n)/2$ is when every pair of names have a similarity of at least a, where $0 \leq a \leq 1$ is the threshold used.

Random naming [7]. [Impair: Transparency, Readability]

This obfuscation renames all identifiers randomly, and results in the loss of meaningful names. Validating that names make sense and reflect their meaning is complicated. Deissenboeck and Pizka [8] identified the importance of proper naming, and created a formal model with naming rules for better more comprehensible code. These rules check consistency, conciseness, and composition of each element name. We suggest using these rules to generate three different metrics that evaluate the code by the validity of its identifier naming.

Let C denote the set of all relevant concepts within a certain scope, N all the used names, and R the relation of names to concepts, $R \subseteq N \times C$. Consistency is the proper relation between names and concepts, and is measured by checking for homonyms and synonyms. A name is a homonym if it has more than one meaning. Different names with the same meaning are considered synonyms. More formally [8], a name $n \in N$ is a homonym iff $|C_n| > 1$ where $C_n = \{c \in C \mid (n,c) \in R\}$. Names n, m are synonyms iff $C_n \cap C_m \neq \emptyset$.

Definition: Let all distinct identifier names be given indices from 1 to $|N|$. Let $ho_i = |C_i|$ be the number of concepts that id_i reflects, and sy_i be the number of synonyms for id_i including itself. Our Consistent Identifiers Metric will be:

$$COIM = \sum_{i=1}^{|N|} (ho_i \cdot sy_i)$$

Scale: $|N| \leq COIM \leq |N|^2|C|$, where $|N|$ is when all the names are consistent, and $|N|^2|C|$ is when all the names are synonyms and each one of them reflects all $|C|$ different concepts. This metric does not only reflect the amount of inconsistent identifiers, it also roughly evaluates the level of inconsistency, by using the multiplication. When one encounters some identifier id_i, one should take into consideration its synonyms as well as all of their possible homonyms.

Conciseness "requires an identifier to have exactly the same name as the concept it stands for" [8]. The way of implementing this is debatable, but it would surly require the use of some domain specific dictionary, together with not having synonyms or homonyms.

Definition: Let $|N|$ be the number of distinct identifier names, and cs_i the conciseness of identifier i, such that $cs_i \in \{0,1\}$. Our Concise Identifiers Metric will be:

$$CSIM = |N| - \sum_{i=1}^{|N|} cs_i$$

Scale: $0 \leq CSIM \leq |N|$, where $|N|$ is when all the names are concise, and 0 is when none of them are.

Deissenboeck and Pizka [8] use the head modifier schema to evaluate the validity of non-atomic names composition, where the concept identified by a compound identifier must be a specialization of the concept identified by its head. For example, the identifier LinkedHashMap is a specialization of the concept HashMap which in turn is a specialization of the concept Map. We propose a metric that would evaluate the number of identifiers that are validly composed. This metric is, off course, intended for identifiers of concepts or objects, that could be composed by the above schema. A different schema may be needed for action-related identifiers such as method and function names.

Definition: Let $|N|$ be the number of distinct identifier names, and M the set of distinct non-atomic identifier names. Let vc_i reflect the validity of identifier i's composition, such that $vc_i \in \{0,1\}$; if id_i is atomic ($id_i \notin M$) then necessarily $vc_i = 1$. The Identifier Composition Validity Metric will be:

$$ICVM = |N| - \sum_{i=1}^{|N|} vc_i$$

Scale: $0 \leq ICVM \leq |M|$, where 0 is when all the non-atomic names have a valid composition, and $|M|$ is when none of them do.

Objectification [3]. [Impair: Transparency, Design]

This simple obfuscation takes a class and changes all of its field types to "object". An "object" type rather than an explicit one has higher potency. We suggest counting the number of such declarations.

Definition: Let T be the set of all variable definitions, and Obj the set of all variables defined with type "object". Then, our Object Metric will be:

$$OBJM = |Obj|$$

Scale: $0 \leq OBJM \leq |T|$, where $|T|$ is when all the variables are defined as "object", and 0 is when none of them are.

Dead code [3]. [Impair: Transparency, Design, Flow complexity]

Dead code insertion is an obfuscation that adds code that is never executed, but is hard to identify as such. For instance, using an opaque predicate (e.g. a hard to understand "if" that actually always evaluates to "false") one can add a block of irrelevant code. But this can also happen in the course of code evolution, e.g. when a function falls out of use. Not all dead code can be detected and eliminated, but Christodorescu and Jha [6] for example proposed a detection tool which can identify several kinds of dead-code segments with acceptable performance. Dead code is known to degrade the efficiency of a program [11], but apparently has not been suggested as a metric for quality and specifically an impediment to comprehension. We suggest that locating and quantifying the relative amount of dead code can be used as a metric.

Definition: Let D_{LOC} be the lines of code which are considered dead and LOC the total lines of code in the program. Then, our Dead Code Metric will be:

$$DCM = \frac{D_{LOC}}{LOC}$$

Scale: $0 \leq DCM \leq 1$, where 1 is when all the code is unreachable, and 0 is when there is no dead code.

Redundant operands [7]. [Impair: Transparency, Readability]

This obfuscation uses algebraic laws to add redundant operands to arithmetic expressions. It increases program length, but the complexity (or number of redirections) of an expression has much more impact than its length. We suggest starting with a simple metric that counts the number of complex expressions, where an expression's complexity is measured by its operator count.

Definition: Let n be the number of all arithmetic expressions. Let ex_i be expression i, and let op_i be the collection of operators in ex_i. If a is the threshold on operators which makes an expression complex, then our Complex Expression Metric will be:

$$CEM = |\{ex_i \mid a < |op_i|\}|$$

Scale: $0 \leq CEM \leq n$, where 0 is when none of the expressions has more the a operators, and n is when all of them do.

The combination of arithmetic and logic in the same expression seems to make an expression even more complicated than simply adding operators. Thus, we also suggest quantifying the appearances of such combinations.

Definition: Let AR be the set of all arithmetical operators, and LO the set of all logical operators. We count the number of expressions which combine arithmetic and logic operators, so our Arithmetic Logic Metric will be:

$$ALM = |\{ex_i \mid LO \cap op_i \neq \emptyset, AR \cap op_i \neq \emptyset\}|$$

Scale: $0 \leq ALM \leq n$, where 0 is when none of the expressions combine arithmetic and logic, and n is when all of them do.

V. Conclusions

Comprehension and obfuscation are two sides of the same coin. Thus techniques developed for code obfuscation can help to identify impediments for code comprehension. And once identified, they can be used to derive new code metrics. However, significant work remains to fill in the details (e.g. select threshold values) and assess the practical usefulness of these metrics. An especially interesting question we plan to address is whether these things indeed occur in real code, and to what degree.

Acknowledgments This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13).

References

[1] F. P. Brooks, Jr., "*No silver bullet: Essence and accidents of software engineering*". *Computer* **20(4)**, pp. 10–19, Apr 1987.
[2] R. P. L. Buse and W. R. Weimer, "*A metric for software readability*". In *Intl. Symp. Softw. Testing & Analysis*, pp. 121–130, Jul 2008.
[3] M. Ceccato, A. Capiluppi, P. Falcarin, and C. Boldyreff, "*A large study on the effect of code obfuscation on the quality of Java code*". *Empirical Softw. Eng.* 2015.
[4] M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "*A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques*". *Empirical Softw. Eng.* **19(4)**, pp. 1040–1074, 2014.
[5] J.-T. Chan and W. Yang, "*Advanced obfuscation techniques for Java bytecode*". *J. Syst. Softw.* **71(1)**, pp. 1–10, 2004.
[6] M. Christodorescu and S. Jha, "*Static analysis of executables to detect malicious patterns*". In *12th USENIX Security Symp.*, pp. 169–186, 2003.
[7] C. Collberg, C. Thomborson, and D. Low, *A taxonomy of obfuscating transformations*. Tech. rep., Dept. Computer Science, University of Auckland, New Zealand, 1997.
[8] F. Deissenboeck and M. Pizka, "*Concise and consistent naming*". *Softw. Quality J.* **14(3)**, pp. 261–282, 2006.
[9] E. W. Dijkstra, "*Go To statement considered harmful*". *Comm. ACM* **11(3)**, pp. 147–148, Mar 1968.
[10] S. Henry and D. Kafura, "*Software structure metrics based on information flow*". *IEEE Trans. Softw. Eng.* **SE-7(5)**, pp. 510–518, Sep 1981.
[11] J. Knoop, O. Rüthing, and B. Steffen, "*Partial dead code elimination*". In *Prog. Lang. Design & Implementation*, pp. 147–158, ACM, 1994.
[12] D. Low, "*Java control flow obfuscation*". MSc Thesis, University of Auckland, Jun 1998.
[13] T. McCabe, "*A complexity measure*". *IEEE Trans. Softw. Eng.* **SE-2(4)**, pp. 308–320, Dec 1976.
[14] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 1999.
[15] D. Posnett, A. Hindle, and P. Devanbu, "*A simpler model of software readability*". In 8th *Working Conf. Mining Softw. Repositories*, pp. 73–82, May 2011.
[16] SonarSource, "*SonarQube*". 2013.
URL `http://www.sonarqube.org/`
[17] M. Sosonkin, G. Naumovich, and N. Memon, "*Obfuscation of design intent in object-oriented applications*". In *Proc. 3rd ACM workshop on Digital Rights Mgmt.*, pp. 142–153, 2003.