# Evaluating Obfuscation Security: A Quantitative Approach

**Conference Paper** *in* Lecture Notes in Computer Science · October 2015

DOI: 10.1007/978-3-319-30303-1_11

CITATIONS
4

READS
614

**2 authors:**

Rabih Mohsen
Imperial College London
**5** PUBLICATIONS **90** CITATIONS
SEE PROFILE

Alexandre Pinto
Royal Holloway, University of London
**19** PUBLICATIONS **66** CITATIONS
SEE PROFILE

# Evaluating Obfuscation Security: A Quantitative Approach

Rabih Mohsen[1] and Alexandre Miranda Pinto[2]

[1] Department of Computing, Imperial College London, UK
[2] Information Security Group, Royal Holloway University of London, UK and Instituto Universitário da Maia, Portugal

**Abstract.** State of the art obfuscation techniques rely on an unproven concept of security, therefore it is very hard to evaluate their protection quality. In previous work we introduced algorithmic information theory as a theoretical foundation for code obfuscation security. We propose Kolmogorov complexity, estimated by compression, as a software complexity metric to measure regularities in obfuscated programs. In this paper we provide a theoretical validation for its soundness as a software metric, so it can have as much credibility as other complexity metrics. Then, we conduct an empirical evaluation for 43 obfuscation techniques, which are applied to 10 Java byte code programs of SPECjvm2008 benchmark suite using three different decompilers as a threat model, aiming to provide experimental evidence that support the formal treatments.

## 1 Introduction

Man-at-the-end (MATE) attacks are performed by an adversary who has physical access to a device or software and can compromise it. Malicious reverse engineering attack, is a typical MATE attack, which violates the confidentiality rights of the vendor by extracting software intellectual property (such as algorithms) or sensitive data (such as license codes or cryptographic keys). Software protection techniques such as code obfuscation, are vital to defend programs against vicious reverse engineering attacks. An obfuscating transformation attempts to manipulate code in such a way that it becomes unintelligible to human and automated program analysis tools, while preserving their functionality.

Collberg et al. [6] were the first to define obfuscation in terms of a semantics-preserving transformation function. Barak et al. [2] provided a formal definition of obfuscation based on virtual black box model, in an attempt to achieve a well-defined security. However, they found a counterexample which showed that this kind of "secure" definition cannot be met. lntuitively, they proved the existence of a set of programs or functions that are impossible to obfuscate. On the other hand, Dalla Preda and Giacobazzi [15] proposed a semantics based approach to define obfuscation security using abstract interpretations theory, aiming to provide a formal tool for comparing obfuscations with respect to their potency. A more recent study conducted by Garg et al. [8] provided promising positive results, using indistinguishability obfuscation, for which there are no known impossibility results.

Our motivation is derived from the difficulty of evaluating the strength of seemingly resilient obfuscating transformations. There is a need to evaluate how obfuscating and de-obfuscating transformations affect the understanding of the program. Many obfuscation

transformation techniques were proposed, which intuitively make the program difficult to understand and harder to attack, with no provable properties presented to measure. Several attempts were made to provide concrete metrics for evaluating obfuscation such as in [6], using classical complexity measures. However, most of these metrics are still context dependent and differ among development platforms, therefore it is very hard to standardise them. This reason and the fact that there are currently no provable security metrics to measure the quality of the code obfuscation, leads to the following questions:

- Is there any theory that can provide an intuitive way to define and explain obfuscation security?
- Can we derive from that theory a quantitative metric, with practical relevance, which can be used to measure the protection level in code obfuscation?
- How to evaluate the usefulness of this metric?

We tried to answer the first question in [14] by providing a theoretical foundation for obfuscation theory based on *algorithmic information theory*, and we proposed a novel metric for code obfuscation that is based on Kolmogorov complexity. In this paper, we aim to answer the remaining questions. First, we apply Weyuker's validation framework [17] to check whether Kolmogorov complexity is theoretically sound as software metric, then we derive a normalised version of Kolmogorov complexity that is approximated by compression. To provide the empirical validation, we conducted an experiment, using the proposed metric, on obfuscated Java jar files of SPECjvm2008 benchmark suite by applying a number of most widely used obfuscation techniques. Specifically, we investigate the quality of obfuscation techniques in two obfuscators: *Sandmark* [3] an open source suite, and *Dasho* [4] a commercial tool. Moreover, we employed three decompilers as a model of attack to study the resilience of code obfuscation.

The theoretical results show that Kolmogorov complexity is theoretically sound with respect to Weyuker's validation framework. The empirical results show that obfuscation techniques managed to produce a substantial increase in the proposed metric comparing to original unobfuscated programs, which confirms our formal definition [14] about code obfuscations. Furthermore, all decompilation attacks demonstrate a different level of success at reducing the complexity of obfuscated programs; however not to the level where it matches the complexity of original unobfuscated programs. We also compared our results, in particular, Sandmark obfuscation techniques with the recent results of Ceccato et al. [4] using Cyclomatic complexity measure [13]. We find that our metric is more sensitive then Cyclomatic measure at detecting any increase in complexity comparing to original unobfuscated code.

The remainder of this paper is structured as follows. In Section 2, we provide an overview of related work. Section 3 provides the preliminaries and background theory about Kolmogorov complexity. In Section 4 we provide the motivation behind our approach. Section 5 presents the theoretical validation, and proposes a normalised version of Kolmogorov complexity as metric to measure code obfuscation. Section 6 describes the experimental design, and discusses the experimental results. Finally, Section 7 concludes and provides the future work.

[3] `http://sandmark.cs.arizona.edu`

[4] `http://www.preemptive.com/products/dasho`

## 2 Related work

The first attempt to evaluate obfuscation was conducted by Collberg et al. [6]; they relied on classical software complexity metrics to evaluate obfuscation such as Cyclomatic Complexity, and Nesting Complexity. Anckaert et al. [1] suggested a framework of four program properties that reflect the quality of code obfuscation: code, control flow, data and data flow, then they applied software complexity metrics to measure these properties; however they did not perform any validation on the proposed metrics. Ceccato et al. [5] experimentally assessed one obfuscation technique (identifier renaming) using statistical reasoning. They measured the success and the efficiency of an attacker by considering the human factor in their threat model, without introducing any new metrics. In a recent study by Ceccato et al. [4] a set of software metrics (modularity, size and complexity of code) were applied to a set of obfuscated programs to measure their effectiveness. Their results show that a limited number of obfuscated techniques, involved in their study, were effective in making code metrics change substantially from original to obfuscated code, in Section 6.2 we discuss the similarity with our results.

Jbara et al. [10], argued that the most of complexity metrics are syntactic features that ignore the programs global structure. Program global structure may have effect on program understanding, they suggested the use of code regularity that is estimated by compression to measure program comprehension. They conducted a controlled experiment using cognitive tasks on a set of program functions. The results established a positive relation between code regularity and program comprehension. The code regularity, according to Jbara et al., is estimated by compression, which is also used to approximate Kolmogorov complexity [11]. Their intuitions and results agree with our observation and theoretical treatments in [14] for code obfuscation. However, our work differs from their work in two ways: we provide a sound theoretical foundation and validation based on algorithmic information theory ( Kolmorgorov complexity) for code regularity, and justify its use in code obfuscation security. On the other hand, they only used compression to measure code comprehension in empirical sense, without applying any theoretical validation. Secondly, we conducted an experiment on a set of obfuscated programs, hoping to provide empirical evidence that support our theoretical work, whereas they did not apply their experiment to study the effect of compression on obfuscated code.

## 3 Preliminaries and Notations

We use $U$ as the shorthand for a universal Turing machine, $x$ for a finite-length binary string and $|x|$ its length. We use the notation $O(1)$ for a constant, $p(n)$ for a polynomial function with input $n \in \mathbb{N}$. Symbol $\|$ is used to denote the concatenation between two programs or strings. $\mathcal{P}$ is a set of binary programs and $\mathcal{P}'$ is a set of binary obfuscated programs, and $\mathcal{L} = \{\lambda_n \ : \ \lambda_n \in \{0,1\}^+, n \in \mathbb{N}\}$ is a binary set of security (secret) parameters that is used in obfuscation process. Given two sets, $I$ an input set and $O$ an output set, a program functionality (meaning) is a function $[\![.]\!] : \mathcal{P} \times I \rightarrow O$ that computes the program's output given an input and terminates.

### 3.1 Kolmogorov Complexity

Kolmogorov complexity (also known as Algorithmic complexity) is used to describe the complexity or the degree of randomness of a binary string. It was independently

developed by A.N. Kolmogorov, R. Solomonoff , and G. Chaitin in the late 1960s [12]. Intuitively, Kolmogorov complexity of a binary string $p$ is the length of the shortest binary program that describes $p$, and is computed on a Universal Turing Machine. We consider only the prefix version of Kolmogorov complexity (prefix algorithmic complexity) which is denoted by $K(.)$. *Complexity* and *Kolmogorov complexity* terms are sometimes used interchangeably; for more details on prefix Kolmogorov complexity and algorithmic information theory, we refer the reader to [12]. The necessary parts of this theory are briefly presented in the following.

**Definition 1.** ([12]) Let $U(P)$ denote the output of $U$ when presented with a program $P \in \{0,1\}^+$.

1. The *Kolmogorov complexity* $K(x)$ of a binary string $x$ is defined as: $K(x) = \min\{|P| : U(P) = x\}$.
2. The Conditional Kolmogorov Complexity relative to $y$ is defined as: $K(x\,|\,y) = \min\{|P| : U(P,y) = x\}$.

**Definition 2.** ([12]) Mutual algorithmic information of two binary programs $x$ and $y$ is given by : $I_K(x;y) = K(y) - K(y\,|\,x)$.

**Theorem 1 (chain rule [12]).** *For all x, y* $\in \mathbb{N}$

1. $K(x;y) = K(x) + K(y\,|\,x) + O(\log K(x;y))$.
2. $K(x) - K(x\,|\,y) = K(y) - K(y\,|\,x)$ *i.e.* $I_K(x;y) = I_K(y;x)$, *up to an additive term* $O(\log K(x;y))$.

**Theorem 2 ([12]).** *There is a constant c such that for all* $x$ *and* $y$

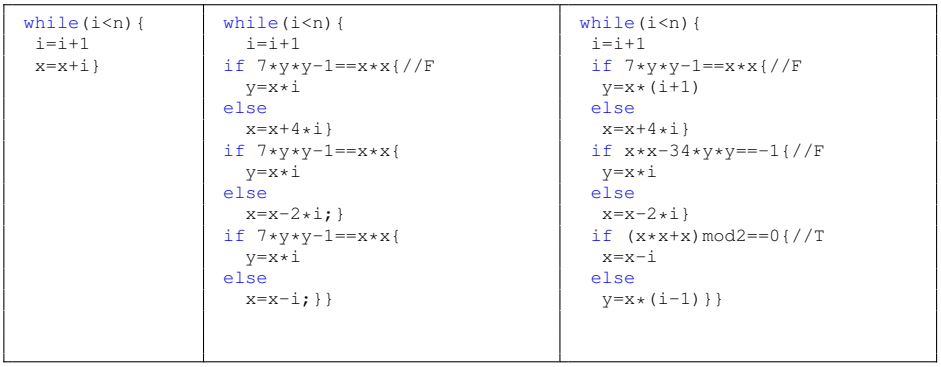
$$K(x) \leq |x| + 2\log|x| + c \;\; \textit{and} \;\; K(x\,|\,y) \leq K(x) + c.$$

Kolmogorov complexity is uncomputable due to the undecidability halting program, however it can be approximated based on compression as shown in [11], Theorem 2 and in [12], this helps to intuitively understand this notion and makes this theory relevant for real world applications. The theorem states that $K(x)$ is the lower bound [11] of all the losseless compressions of $x$; therefore, we say that every compression $C(x)$ of $x$ gives an estimation of $K(x)$.

## 4 Obfuscation using Kolmogorov Complexity

The main purpose of code obfuscation is to confuse an adversary, making the task of reverse engineering extremely difficult. Code obfuscation introduces noise and dummy instructions that produce irregularities in the targeted obfuscated code. We believe that these make the code difficult to comprehend, that is, obfuscated. Classical complexity metrics have a limited power for measuring and quantifying irregularities in obfuscated code, because most of these metrics are designed to measure certain aspects of code attributes such as finding bugs and code maintenance. Code regularity (and irregularity) can be quantified, as was suggested in [10], using Kolmogorov complexity and compression. Code regularity means a certain structure is repeated many times, and thus can be recognized. Conversely, irregularities in code can be explained as the code exhibiting different types of structure over the code's body.

The main intuition behind our approach is based on the following argument: if an adversary fails to capture some patterns (regularities) in an obfuscated code, then the adversary will have difficulty comprehending that code: it cannot provide a valid, brief, or simple description. On the other hand, if these regularities are simple to explain, then describing them becomes easier, and consequently the code will not be difficult to understand.



```
while(i<n){
 i=i+1
 x=x+i}
```

(a) Sum code

```
while(i<n){
  i=i+1
if 7*y*y-1==x*x{//F
  y=x*i
else
  x=x+4*i}
if 7*y*y-1==x*x{
  y=x*i
else
  x=x-2*i;}
if 7*y*y-1==x*x{
  y=x*i
else
  x=x-i;}}
```

(b) One opaque predicate

```
while(i<n){
 i=i+1
 if 7*y*y-1==x*x{//F
  y=x*(i+1)
 else
  x=x+4*i}
 if x*x-34*y*y==-1{//F
  y=x*i
 else
  x=x-2*i}
 if (x*x+x)mod2==0{//T
  x=x-i
 else
  y=x*(i-1)}}
```

(c) Three opaque predicate

**Fig. 1.** Obfuscation example: (a) is the original code for the sum of $n$ integers; (b) is an obfuscated version of (a) with one opaque predicate and data encoding which has some patterns and regularities; (c) is another obfuscated version of (a) with three opaque predicate and data encoding, which has less patterns and regularities comparing to (b).

**Example:** We demonstrate our motivation using the example in Fig. 1. We obfuscate the program in Fig. 1-(a), which calculates the sum of the first $n$ positive integers, by adding opaque predicates[5] with bogus code and data encoding. If we apply Cyclomatic complexity, a classical complexity measure, to Fig. 1-(b) the result will be 7. Cyclomatic complexity is based on control flow graph (CFG), and is computed by: $E - N + 2$, where $E$ is the number of edges and $N$ is the number of nodes in CFG. Fig. 1-(b) contains $N = 8$ nodes, $E = 13$ edges then the Cyclomatic complexity is $(13 - 8 + 2) = 7$. We can see some regularity here: there is one opaque predicate repeated three times. Furthermore, the variable $y$ is repeated three times in the same place of the `If`-branch.

We take another obfuscated version in Fig. 1-(c) (of the same program); this code is obfuscated by adding three different opaque predicates. The patterns and regularities becomes less in this version comparing to Fig. 1-(b); however the Cyclomatic complexity is still the same 7, and it does not account for the changes that occurred in the code. Assuming the opaque predicates of Fig. 1-(c) are equally difficult to break, attacking this code requires at least twice more effort than the code in Fig. 1-(b), as we need to figure out the value of two more opaque predicates.

Furthermore, Fig. 1-(b) can be compressed at higher rate than Fig. 1-(c); again, this is due to the inherent regularity in Fig. 1-(b). We argue that an obfuscated program which is secure and confuses an adversary will exhibit a high level of irregularity in its source

[5] An opaque predicate is an algebraic expression which always evaluates to same value (true or false) regardless of the input.

code, and thus it requires a longer description to characterize all its features. This can be captured by the notion of Kolmogorov complexity, which quantifies the amount of information in an object.

### 4.1 Applying Kolmogorov Complexity to Code Obfuscation

So far, we argued that Kolmogorov complexity can be used to determine regularities in code obfuscation. The question that makes sense to ask is: At which level of Kolmogorov complexity can we claim an obfuscated code is secure against a specific adversary? It is vital to provide first some sort of formal definition of code obfuscation, that captures the desired security properties. Although this task is quite difficult and cumbersome, we provided an intuitive definition in [14] that is inspired by practical uses of obfuscation. We believe that it is the first step toward establishing a solid foundation for code obfuscation. The rationale behind this definition is that an obfuscated program must be more difficult to understand than the original program. This uses the notion of $c$-unintelligibility:

**Definition 3.** (Unintelligibility)([14]) A program $P'$ is said to be $c$-unintelligible with respect to another program $P$ if it is $c$ times more complex than $P$, i.e. the added complexity is $c$ times the original one, and thus more difficult to understand. Formally: $K(P') \geq (c+1)K(P)$, for some constant $c > 0$.

**Definition 4.** A *c-Obfuscator* $\mathcal{O} : \mathcal{P} \times \mathcal{L} \rightarrow \mathcal{P}'$ is a mapping from programs with security parameters $\mathcal{L}$ to their obfuscated versions such that $\forall P \in \mathcal{P}, \forall \lambda \in \mathcal{L}$ . $\mathcal{O}(P, \lambda) \neq P$ and satisfies the following properties:

- **Functionality**: $O(P, \lambda)$ and $P$ compute the same function, such that $\forall i \in I$. $[\![P]\!](i) = [\![\mathcal{O}(P, \lambda)]\!](i)$.
- **Polynomial Slowdown**: the size and running time of $\mathcal{O}(P, \lambda)$ are at most polynomially larger than the size and running time of $P$.
- **Unintelligibility**: $\mathcal{O}(P, \lambda)$ is $c$-unintelligible with respect to $P$.

We could use the unintelligibility property to answer the aforementioned question. If we need to secure a program against an adversary, whether it is human or an automatic reverse engineering tool, we may require an obfuscator to add $c$ amount of irregularities that could foil that adversary.

### 4.2 Security Model

To properly define security we need to specify the capabilities of our attacker. The most basic case we are trying to capture is that of a human who seeks to obtain some original code from an obfuscated version of it, without the assistance of any automated tools. The difficulty of the analyst's task is measured by the amount of information that s/he lacks to obtain the target code. If the obfuscation is weak, this will be small. A good obfuscation will force the analyst to obtain more information to reach its target, possibly some of the randomness used to execute the obfuscation in the first place.

At the other extreme, we have an analyst with access to the complete range of analysis tools. It can compute any function[6] of the obfuscated code, and eventually produce a

[6] Any *computable* function, that is.

modified version thereof. Ultimately, it will seek to produce a deobfuscated version of the program, that is, well-structured code which is similar to the original program and has less noise. In this set of functions, we include, for example, automated reverse-engineering analysis techniques such as static program analysis (e.g., data flow, control flow, alias analysis, program slicing, disassemblers, and decompilers) and dynamic program analysis (e.g., dynamic testing, profiling, and program tracing).

While the first adversary is too limited to be realistic, the powers of the second are too broad to be useful. Every obfuscated program that is obtained by a deterministic transformation (taking clear code and auxiliary randomness as input) can be reversed by undoing the exact same steps in the opposite order. Therefore, for each instance there will be at least one computable function that is able to return the original code. To achieve a meaningful model we have to establish limits to the adversary power and exclude such situations that would make any obfuscation trivially impossible.

We do this by letting the adversary run functions of the obfuscated code, but only from a list chosen before it receives the challenge obfuscation (i.e., independent from the challenge). Besides, adversaries are parameterized by the amount of information they use: this includes their own code, that of any algorithm they choose to run and any other auxiliary information. The adversary wins if it produces a candidate deobfuscation that is close to the original and this does not require too much information. Formally, for a security parameter $0 \leq \epsilon \leq 1$, the adversary wins if:

- It receives obfuscated code $P' = \mathcal{O}(P, \lambda)$ and produces deobfuscation $P^*$
- $P^* = P$, and the Kolmogorov complexity of the adversary's information, $Q$, is less than $(1 - \epsilon)K(P)$.

This definition is compatible with the definition of security in [14]: if the last condition is true, the adversary successfully produce $P$ from $P^*$ using information $Q$. Therefore, $K(P|P') \leq K(Q) < (1 - \epsilon)K(P)$ and so $I_K(P; P') > \epsilon K(P)$.

## 5 Theoretical Metric Validation

Obfuscated programs are software in the first place. Measuring obfuscation means we are quantifying some software properties that may reflect the code security. Although the security property is captured using unintelligibility according to the above definition, Kolmogorov complexity requires validation to ensure its acceptance, usefulness and soundness as a software metric. Theoretical validation is considered as a necessary step before empirical validation. Several properties have been suggested for theoretical validating of software complexity measures such as Weyuker [17] and Briand et al. [3]. Among the proposed models, we found Weyuker's axioms are more suitable for validating Kolmogorov complexity. Weyuker's validation properties, despite the criticisms that were received [16], have been broadly applied to certify many complexity measures, and are still an important basis and general approach to certify a complexity measure. Weyuker proposed nine properties or axioms for complexity validation, which we use to validate Kolmogorov complexity. There are some concepts presented in Weyuker's properties that require some clarification in the context of Kolmogorov complexity such as functional equivalence, composition of two programs, permutation of statements order and renaming. Consider two programs $P$ and $Q$ that belong to a set of binary strings $\{0, 1\}^+$.

– Functional equivalence: $P$ and $Q$ are said to have the same functionality if they are semantically equivalent i.e. given identical input, the output of the two programs are the same, i.e. $\forall i \in I.[\![P]\!](i) = [\![Q]\!](i)$.
– Composition : Although Weyuker did not include any formal relation to identify the composition of two programs, we consider the composition in the context of Kolmogorov complexity as the joint Kolmogorov complexity, which can be expressed as the concatenation of $P$ and $Q$ programs before applying the complexity measure. $K(P;Q) = K(P \parallel Q)$ where $\parallel$ is the concatenation between $P$ and $Q$.
– Permutation: A program $P \in \{0,1\}^+$ can be composed of concatenated sub-binary strings $p_i \subset P$, for example it may represent program instructions, such that: $P = p_1 \parallel ... \parallel p_n$. The permutation involves changes in the order or the structure of how these binary substrings are represented in $P$.
– Renaming: Renaming refers to syntactic modification of a program's identifiers, variables and modules names.

Weyuker validation properties are presented in the following, where $C_o$ is a complexity measure that maps a program to a non-negative number.

**Definition 5.** [[17]] (Weyuker's validation properties) A complexity measure $C_o : P \rightarrow \mathbb{R}$ is a mapping from a program to a non-negative real number and has the following properties:

1. **Not constant:** $\exists P, Q.\ C_o(P) \neq C_o(Q)$. This property states the complexity measure *is not constant*.
2. **Non-coarse:** Given a nonnegative number $c$, there are only a finite number of programs such that $C_o(P) = c$.
3. **Non-uniqueness:** $\exists P, Q.\ P \neq Q \wedge C_o(P) = C_o(Q)$. This property again states that the measure is nontrivial, in that there are multiple programs of the same size.
4. **Functionality:** $\exists P, Q.\ \forall i \in I.[\![P]\!](i) = [\![Q]\!](i)) \wedge C_o(P) \neq C_o(Q)$. It expresses that there are functionally equivalent programs with different complexities.
5. **Monotonicity:** $\forall P, Q.\ C_o(P) \leq C_o(P \parallel Q) \wedge C_o(Q) \leq C_o(P \parallel Q)$. This property checks for monotonic measures. It states that adding to a program makes it increase its complexity.
6. **Interaction matters (a):** $\exists P, Q, R.\ C_o(P) = C_o(Q) \wedge C_o(P \parallel R) \neq C_o(Q \parallel R)$. This property explains the interaction of two equal complexity programs with an auxiliary concatenated program. It states that a program $R$ may produce different complexity measure when it is added to two equal complexity programs $P$ and $Q$.
   **Interaction matters (b):** $\exists P, Q, R.\ C_o(P) = C_o(Q) \wedge\ C_o(R \parallel P) \neq C_o(R \parallel Q)$. This property is similar to the previous except that the identical code occurs at the beginning of the program.
7. **Permutation is significant:** Let $\pi(P)$ be a permutation of $P$'s statements order. Then, $\exists P.\ C_o(P) \neq C_o(\pi(P))$. This expresses that changing the order of statements may change the complexity of the program.
8. **Renaming:** If $P$ is a renaming of $Q$, $P = Rename(Q)$, then $C_o(P) = C_o(Q)$. This property asserts that uniformly renaming variable names should not change a program's complexity.

9. **Interaction may increase complexity:** $\exists P, Q.\ C_o(P) + C_o(Q) \leq C_o(P \parallel Q)$. This property states that a merged program of two programs can be more complex than its component parts.

The Renaming property as was suggest by Weyuker is not desirable for code obfuscation. Functionally it is true, a renaming of variables does not in any way alter the structure of the code. However, it is easy to see that it can make human understanding much more difficult. A good programming practice is to use clear names for variables and methods, that explain accurately what they do and go a long way towards reducing the necessity of comments in the code. Conversely, long random names obscure their meaning forcing the analyst to follow the program's logic to understand their functionality. From a Kolmogorov point of view, meaningful names have a smaller complexity than long ones, and a program with renamed variables might well be more complex which suits our intuitition regarding its comprehensibility. We consider this proposition no further.

Weyuker argued that property 9 helps to account for a situation that a program's complexity increases as more additional components are introduced, due to the potential interaction among these parts. Briand et al. [3] provided a modified version of this property (stronger version) called **Disjoint Module Additivity**, which establishes a relation between a program and the complexity of its parts. Given two disjoint modules $m_1$, $m_2$ such that $P = m_1 \cup m_2$ and $m_1 \cap m_2 = \varnothing$ where $P$ is the whole program, then $C_o(m_1) + C_o(m_2) = C_o(P)$. Below we check whether these properties are satisfied by Kolmogorov complexity.

**Proposition 1 (Not constant).** $\exists P, Q.K(P) \neq K(Q)$.

*Proof.* By simple counting, there are at most $2^n$ programs with complexity at most $n$. Therefore, there must be programs with complexity larger than $n$ and so there must be programs with distinct complexities.

**Proposition 2 (Non-coarse).** *Given a nonnegative number $c$, there are only a finite number of programs such that* $\exists d.|\{P : \exists c.K(P) = c\}| \leq d$.

*Proof.* According to Theorem 7.2.4 in [7] the number of strings (in our context programs) of Kolmogorov complexity less than or equal to $k$ is upper bounded by $2^k$, i.e. $|S = \{\forall P \in \{0,1\}^* : K(P) \leq k\}| \leq 2^k$, which means set $S$ is finite.

**Proposition 3 (Non-uniqueness).** $\exists P, Q.P \neq Q \wedge K(P) = K(Q)$.

*Proof.* It is possible to construct a prefix-free code with $2^n$ strings of length up to $n$, e.g. the code composed only of all the strings of length $n$. By basic properties of Kolmogorov complexity, these have complexity up to $n + O(1)$. If the proposition is false, there is at most one minimal program of each size. Therefore, there could be at most $n - 1$ non-empty strings of length smaller than $n + O(1)$ with complexity up to $n + O(1)$, which is a contradiction. Thus, there must be strings with the same complexity.

**Proposition 4 (Functionality).** $\exists P, Q.(\forall i \in I.[\![P]\!](i) = [\![Q]\!](i)) \wedge K(P) \neq K(Q)$.

*Proof.* In general, one same function can be produced by several different implementations, that might bear little resemblance (e.g. different sorting algorithms, all producing the same result). Therefore, in general their complexities will be different. For an extreme example, consider program $P$ and let $Q \parallel R$, where $R$ is an added program that does not touch on any of the variables, memory or other resources of $P$ and does not return results. It takes resources and does work, but ultimately $Q$ just returns what $P$ returns. Then, $\forall i \in I.[\![P]\!](i) = [\![Q]\!](i)$ and $K(Q) = K(P) + K(Q|P) \geq K(P)$. And because $Q$ has to be independent from $P$ in order to use other resources, it must be that $K(Q|P) = K(Q)$ and the inequality is strict.

**Proposition 5 (Monotonicity).** $\forall P, Q.\ K(P) \leq K(P \parallel Q)\ \wedge\ K(Q) \leq K(P \parallel Q)$.

*Proof.* We need to prove that $K(P \parallel Q)$ is greater than $K(P)$ and $K(Q)$. By Theorem 1, $K(P, Q) = K(P) + K(Q|P) + c$ where $c$ is an additive constant. Up to a logarithic term, $K(P \parallel Q) = K(P, Q) = K(P) + K(Q|P)$. By definition, $K(Q|P) \geq 0$ and so $K(P) \leq K(P \parallel Q)$. The proof is equal for $K(Q)$.

**Proposition 6 (Interaction matters).** *(a)* $\exists P, Q, R.\ K(P) = K(Q) \wedge K(P \parallel R) \neq K(Q \parallel R)$ *and (b)* $\exists P, Q, R.\ K(P) = K(Q) \wedge K(R \parallel P) \neq K(R \parallel Q)$.

*Proof.* Assume the existence of two binary programs $P, Q$ such that $K(P) = K(Q)$ and $I(P, Q) = O(1)$. Let $R = P$. Then, we have that up to small approximation factors $K(P \parallel R) = K(P, R) = K(P) + K(R|P) = K(P) + O(1)$. On the other hand, $K(Q \parallel R) = K(Q, R) = K(Q) + K(R|Q) = K(P) + K(R)$ where the last equality follows from the definition of mutual information. If $R$ must be different than either $P, Q$, then we can repeat the same proof by picking a program $R$ that has high $I(P, R)$ but small $I(Q, R)$, for example, a truncation of $P$.

**Proposition 7 (Permutation).** *Given a permutation* $\pi$, $\exists P.\ K(P) \neq K(\pi(P))$.

*Proof.* Fix a program $P$ with $n$ distinct lines, each at most $m$ bits long. There are $n!$ permutations of $P$, and because the lines are all distinct these lead to $n!$ different permuted programs. We show that there must be a program $Q$ corresponding to some permutation $Q = \pi(P)$ such that $K(Q) > K(P)$. By construction, $|P| = mn$ and so there are at most $2^{mn}$ strings with complexity smaller or equal to $P$. By Stirling's approximation, $\ln n! = n \ln n - n + O(\ln n)$. Pick $n$ such that $\ln n > \ln(2) \cdot m + 1$, which implies $n \ln n - n > \ln(2) \cdot mn \Rightarrow n! > 2^{mn}$. Then, there are more permuted programs that strings less complex than $P$ and so at least one permutation leads to a program more complex than $P$.

**Proposition 8 (Disjoint Module Additivity).** $\exists P, Q.K(P) + K(Q) = K(P \parallel Q)$.

*Proof.* $K(P; Q) = K(P) + K(Q|P)$ by Theorem 1. Assume $P \cap Q = \varnothing$, then $K(Q|P) = K(Q)$ since the two programs are fully independent; therefore $K(P; Q) = K(P \parallel Q) = K(P) + K(Q)$ up to logarithmic precision.

The above results show that Kolmogorov complexity satisfies all Weyuker's properties in definition Definition 5, with two weak exceptions that have been addressed above. Therefore, we conclude Kolmogorov complexity is a suitable complexity measure for software based on Weyuker's validation framework.

### 5.1 Normalized Kolmogorov Complexity

Kolmogorov Complexity is an absolute measure, which is problematic when we want to compare two programs with different sizes. For example consider a program $P$ of 1000 bits size that can be compressed to 500 bits, take another program $Q$ of $10^6$ bits size, which is compressed to 1000 bits. By using the absolute measure of Kolmogorov complexity, $Q$ is more complex than $P$. However, $P$ can be compressed to almost half of its size, where $Q$ can be compressed to $\frac{1}{1000}$ of its size, which clearly indicates that $Q$ has more regularities than $P$, and hence that makes $P$ more complex than $Q$. In order to overcome this issue, we suggest a normalized version of Kolmogorov Complexity that is relativized by the upper bound of Kolmogorov complexity i.e. the maximum complexity a certain obfuscated code can achieve. Kolmogorov complexity is upper bounded by the length of its program, the subject of measure, according to Theorem 2; this bound can be used as the maximum Kolmogorov complexity. Normalized Kolmogorov complexity can be useful demonstrating the divergence of obfuscated code complexity before and after a given attack, in terms of information content (high variability of text content), from the maximum value of that complexity.

**Definition 6.** The normalised Kolmogorov complexity $NK$ of a program $P$ is given by:

$$NK(P) = \frac{K(P)}{|P| + 2\log(|P|)}$$

Where $|P|$ is the length of $P$

A high value of $NK$ means that there is a high variability of program content structure, i.e. high complexity. A low value of $NK$ means high redundancy, i.e. the ratio of repeating fragments, operators and operands in code. Since Kolmogorov complexity can be effectively approximated by compression ([11]), it is possible to estimate $NK$ by using a standard compressor instead of a minimal program:

$$NC(P) = \frac{C(P)}{|P| + 2\log(|P|)},$$

implying $0 \leq NC \leq 1$.

We can say that if the size of compressed obfuscated code is equal to the size of obfuscated code itself, then the obfuscated code is considered highly random, and is difficult to comprehend by an attacker. This can be justified in light of our discussion of code regularity in Section 4.

## 6 Experiment Evaluation

The recent empirical result [10] shows that data compression is a promising software metric technique to estimate human comprehension in software. So far, we present a theoretical validation that complements this result, and we propose formal definition that reflects the natural intuition of code obfuscation unintelligibility. Therefore, it is necessary to validate this formal treatment empirically, and to check whether code obfuscation techniques increase the complexity (*NC* measure) of software. Furthermore, we need to examine the extent to which current obfuscation techniques resist reverse engineering tools (decompilers) i.e. prevent reducing the complexity of obfuscated code. Specifically, we aim to answer the following questions:

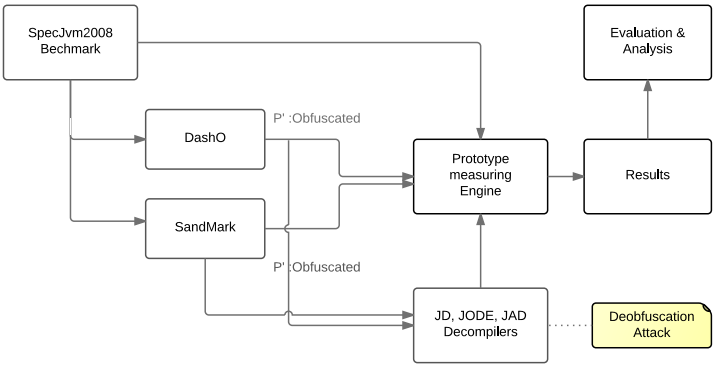



**Fig. 2.** High level overview of the experimental procedure

- What is the effectiveness of obfuscation algorithms using the *NC* measure, by type: control flow, data and layout obfuscation?
- Is there any change in *NC* measure between a clear code and its obfuscated version using different obfuscation algorithms? Does that change, if it occurs, imply an increase in *NC* measure?
- What is the impact of deobfuscation (decompilers) on code obfuscation resilience?

We answered the above questions by conducting an experiment on an obfuscated version of SPECjvm2008 (Java Virtual Machine Benchmark) programs. We obfuscated 10 real-world applications of SPECjvm2008 benchmark suite, ranging in size from medium to large, and containing several real life applications and benchmarks, focusing on core Java functionality. Each one was written in the Java source language and compiled with `javac` to Java byte code, where the obfuscation took place on this level. A brief description of SPECjvm2008 is given in Table 1, the full description and documentation of SPECjvm2008 suite can be found on the benchmark's webpage[7]. The complete list of the obfuscation algorithms that were used in our experiment is provided in Table 2, the full description of each obfuscation techniques can be found on their website.

| **Benchmark** | **Description** |
|---|---|
| compiler | A java decompiler using the OpenJDK (JDK 7 alpha) front end compiler |
| compress | Compresses data, using a modified Lempel-Ziv method (LZW) |
| crypto | Provides three different ciphers (AES,RSA, signverify) to encrypt data |
| derby | An open-source database written in pure Java |
| mpegaudio | MPEG-3 audio stream decoder |
| scimark | A floating point benchmark |
| serial | serializes and deserializes primitives and objects, using data from the JBoss benchmark |
| startup | Starts each benchmark for one operation |
| Sunflow | Tests graphics visualization using multi-threaded global illumination rendering system |
| xml | Has two sub-benchmarks: XML.transform and XML.validation. |

**Table 1.** SPECjvm2008 benchmark brief description

[7] `http://www.spec.org/jvm2008/`

We select two obfuscators of the most prominent tools: one commercial *DashO evaluation copy* with all main features turned on, and a free source version of *SandMark* (see Table 2). The original benchmark `jar` files were obfuscated by using 43 different obfuscation techniques of *DashO* and *Sandmark* obfuscators. We apply three Java decompilers to investigate the resilience, and assess to which extent the applied obfuscation techniques can resist decompilation attacks. Our choice was based on a study by Hamilton and Danicic [9], who investigated the effectiveness of Java decompilers using an empirical evaluation on a group of currently available Java bytecode decompilers. We selected, based on that experiment, three Java decompilers that score the best among all the decompilers in terms of effectiveness and correctness: `JD`[8], `JAD`[9] and `JODE`[10]. We applied *bzip2* compressor to compute *NC*, which is one of the most effective lossless compressor to approximate Kolmogorov complexity, according to [12].

We automate the whole testing proposed, using a scripting code that was written in python to glue the command line versions of Sandmark, Dasho obfuscation, decompilation, and our proposed metric. All of the above components are integrated into our prototype, Fig. 2 shows an overview of the tool-set and the experimental procedure.

### 6.1 Results and Analysis

The results are reported in forms of charts. Fig. 3 shows the results of *NC* according to obfuscation transformation algorithms and decompilation. The bottom line of Fig. 3 with error bars represents the baseline measurement of SPECjvm2008 benchmark before obfuscation process taking place, the error bars are the standard deviation of *NC* among different programs of SPECjvm2008. We used the baseline as a reference comparison line to other results, for example to show if any changes in *NC* occur due to obfuscation techniques and decompilation. To facilitate reading Fig. 3 , we produced artificial gaps among the different obfuscation techniques groups. For convenience presenting the obtained results, we clustered all the obfuscation transformation algorithms into three types of transformations: *Control-flow*, *Data* and *Layout*, see Table 2 and Fig. 4, then we report the average *NC* for each type transformation over the benchmark programs.

**The Impact of Obfuscation on Complexity**

In this section we answered the first two empirical questions. First we investigate the obfuscation effectiveness: Around 50% of obfuscation techniques, of different obfuscation types, have scored similar complexity values (*NC=0.91*) with very minor difference, all of these techniques are part of Sandmark obfuscator framework. This could indicate a common design pattern among these techniques, which needs further investigation. Dasho's obfuscation techniques show a different behavior, only data obfuscation type have a similar complexity values. We also found that *StringEncoder* and *BuggyCode* performed better than all the obfuscation techniques in terms of *NC*, where *ClassSplitter* and *RenameFlattenHierarchyOverInduction* scored the lowest among all obfuscation techniques. Aggregating the obfuscation techniques according to their types, shows a

[8] `http://java.decompiler.free.fr`
[9] `http//varaneckas.com/jad/`
[10] `http://jode.sourceforge.net/`

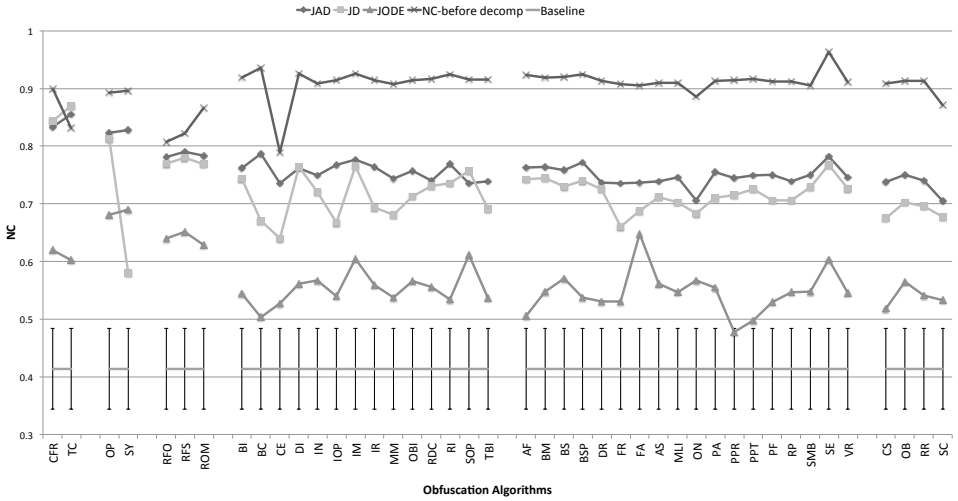



**Fig. 3.** NC measure per obfuscation technique

| Obfuscator | obfuscated technique | Abbr | Obfuscator | obfuscated technique | Abbr |
|---|---|---|---|---|---|
| Dasho-(C) | DControlFlow | CFR | Sandmark-(D) | BlockMarker | BM |
| Dasho-(C) | tryCatch10 | TC | Sandmark-(D) | BludgeonSignature | BS |
| Dasho-(D) | optimisation | OP | Sandmark-(D) | BooleanSplitter | BSP |
| Dasho-(D) | synthetic | SY | Sandmark-(D) | DuplicateRegister | DR |
| Dasho-(L) | FlattenHierarchyOverInduction | RFO | Sandmark-(D) | FalseRefactor | FR |
| Dasho-(L) | FlattenhierarchySimple | RFS | Sandmark-(D) | FieldAssignment | FA |
| Dasho-(L) | OverInductionMaintainhierarchy | ROM | Sandmark-(D) | IntegerArraySplitter | AS |
| Sandmark-(C) | BranchInverter | BI | Sandmark-(D) | MergeLocalIntegers | MLI |
| Sandmark-(C) | BuggyCode | BC | Sandmark-(D) | OverloadNames | ON |
| Sandmark-(C) | ClassSplitter | CE | Sandmark-(D) | ParamAlias | PA |
| Sandmark-(C) | DynamicInliner | DI | Sandmark-(D) | PromotePrimitiveRegisters | PPR |
| Sandmark-(C) | Inliner | IN | Sandmark-(D) | PromotePrimitiveTypes | PPT |
| Sandmark-(C) | InsertOpaquePredicates | IOP | Sandmark-(D) | PublicizeFields | PF |
| Sandmark-(C) | InterleaveMethods | IM | Sandmark-(D) | ReorderParameters | RP |
| Sandmark-(C) | Irreducibility | IR | Sandmark-(D) | StaticMethodBodies | SMB |
| Sandmark-(C) | MethodMerger | MM | Sandmark-(D) | StringEncoder | SE |
| Sandmark-(C) | OpaqueBranchInsertion | OBI | Sandmark-(D) | VariableReassigner | VR |
| Sandmark-(C) | RandomDeadCode | RDC | Sandmark-(L) | ConstantPoolReorder | CS |
| Sandmark-(C) | ReorderInstructions | RI | Sandmark-(L) | Objectify | OB |
| Sandmark-(C) | SimpleOpaquePredicates | SOP | Sandmark-(L) | RenameRegisters | RR |
| Sandmark-(C) | TransparentBranchInsertion | TBI | Sandmark-(L) | SplitClasses | SC |
| Sandmark-(C) | ArrayFolder | AF | | | |

**Table 2.** Obfuscation techniques and their abbreviations used in the experiment, (C) stands for control flow, (D) Data and (L) Layout obfuscation.

very minor difference using the proposed metric see Fig. 4. Data obfuscation and control flow obfuscation performed roughly the same. That was a bit surprising, as we expected the data obfuscation to outperform control-flow obfuscation. This is due to the nature of data obfuscation that adds a lot of noise to program data structure comparing to control flow obfuscation, which only complicates the structure of Control Flow Graph (CFG).

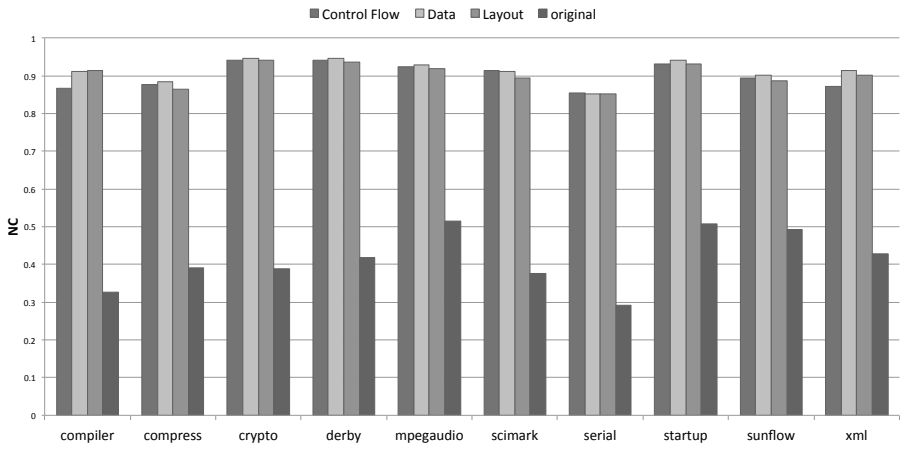



**Fig. 4.** Averaged NC measure per obfuscation transformation. C: Control Flow, D: Data and L: Layout obfuscation

We used the results in Fig. 3 and Fig. 4 to investigate whether code obfuscation changes the complexity of original unobfuscated code. For visual inspection, we see a change in *NC* over all the obfuscation transformation algorithms comparing to the unobfuscated benchmark programs' scores. In Fig. 3, we notice a substantial increase in *NC* all over the obfuscation algorithms comparing to the baseline. We also see a clear increase in complexity of obfuscations per type, see Fig. 4, comparing to the unobfuscated code for all the benchmark programs. These results provide an answer to the second question that there are changes in complexity of software due to obfuscation techniques, and these changes produce a positive increase in the complexity ( *NC*).

### The Impact of Decompilation on Obfuscation Resilience

We answer the last question by investigating the effect of decompilation as an attack on code obfuscation resilience. Fig. 3 reports the results, it shows three different lines labeled with `JAD`, `JD` and `JODE` which resemble the average *NC* of obfuscation techniques after being subjected to decompilation for all the benchmark programs. Most of obfuscation techniques, in this study, shows high resilience against `JD` and `JAD`, and a weak resilience against `JODE`. Analysing each obfuscation technique, we observe a high resilience of Dasho's obfuscation techniques against `JD`, apart of *Synthetic* technique which scores the lowest complexity. *Synthetic* is a technique designed to fail decompilation; however `JD` was very effective at thwarting this technique. *StringEncoder* has the highest resilience against `JD` where *ClassSplitter* demonstrates the lowest resilience. *tryCatch10* shows a high resilience against `JD` and `JAD`, as they are ineffective against the intensive use of `try-catch` blocks. `JODE` performed better than other decompilers at reducing the complexity of obfuscation transformations on individual technique, as we see in Fig. 3. `JODE` was very effective at reducing the complexity of *tryCatch10* obfuscated programs comparing to `JAD` and `JD`. We notice that `JODE` managed to reduce the *NC*

to the benchmark baseline level in Fig. 3. We investigate this matter in more details, and we find the main reason for this decrease in complexity: `JODE` failed to produce a complete decompilation when it decompiles the programs that were obfuscated with arbitrary bytecode, such as *BuggyCode*, we also realized the same problem with `JAD` too. Surprisingly `JODE` failed to replace `java.lang.Integer` object to the correct `int` in the source code for *PromotePrimitiveRegisters*, and *PromotePrimitiveTypes* obfuscation, which agrees with Hamilton et al. [9] observation that `JODE` sometimes fails at resolving and inferring the correct types. Nevertheless, `JODE` decompiled the other obfuscated programs with a reasonable accuracy. In general, all decompilation attacks have managed to reduce *NC* to a certain degree, where `JODE` outperformed all of the decompilers at reducing the complexity of obfuscated programs.

### 6.2 Comparing with Ceccato et al. Results

We compare some of our results, in particular Sandmark obfuscation techniques, with Ceccato et al. [4] results using Cyclomatic complexity measure. In [4], most obfuscations report a Cyclomatic complexity similar to the clear code, only few cases show higher increase in complexity. In our study, we find that most of Sandmark obfuscated programs reported a significant increase in $NC$ complexity.

We further investigated this matter, and observed in Ceccato et al. study that Cyclomatic complexity only accounts for changes in obfuscated programs that involve changes in Control Flow Graph (CFG), i.e. adding or changing basic blocks or edges to CFG, such as some of control flow obfuscation type. We also notice that the obfuscated techniques, which did not report any increase in Cyclomatic complexity, according to Ceccato et al., report a similar *NC* complexity increase in our study. However that was because Cyclomatic measure did not detect any changes in the obfuscation process, whereas *NC* is more sensitive than Cyclomatic measure at detecting any increase in obfuscation complexity comparing to original unobfuscated code.

## 7 Conclusion and Future Work

Compression can be ultimately represented by Kolmogorov complexity; however, this poses the following two questions: How good is Kolmogorov complexity as a metric for code obfuscation? How can we verify the validity of this metric to measure code obfuscation? In previous work [14] we provided a theoretical foundation for the use of Kolmogorov complexity as a metric to measure the security of code obfuscation, without having to rely on classical complexity metrics, and we detailed the intuitions supporting this choice. In this paper we addressed some remaining questions, with efforts divided in two main parts: the first part shows that Kolmogorov complexity is a sound metric to measure software with respect to the properties of Weyuker's validation framework; the second part presents the empirical results which show that obfuscation techniques managed to produce a substantial increase in the proposed metric comparing to original unobfuscated programs, which confirms our formal definition about code obfuscations.

In our experiment, we used an attack model where the adversary can only use static analysis techniques; the metric we proposed does not certify code obfuscations against dynamic analysis tools such as profiling, debugging and dynamic slicers. We plan to extend our framework to certify dynamic code obfuscation as future work.

## References


1. Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. Program bfuscation: a quantitative approach . In *Proceedings of QoP '07*, pages 15–20, New York, USA, October 2007. ACM Press.
2. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *IACR Cryptology ePrint Archive*, 2001:69, 2001.
3. Lionel C. Briand, Sandro Morasca, and Victor R. Basili. Property-based software engineering measurement. *IEEE Trans. Software Eng.*, 22(1):68–86, 1996.
4. Mariano Ceccato, Andrea Capiluppi, Paolo Falcarin, and Cornelia Boldyreff. A large study on the effect of code obfuscation on the quality of java code. *Empirical Software Engineering*, pages 1–39, 2014.
5. Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. The effectiveness of source code obfuscation: An experimental assessment. In *ICPC*, pages 178–187, 2009.
6. Christian Collberg, C. Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations, 1997.
7. Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley, 2006.
8. Sanjam Garg, Mariana Raykova, Craig Gentry, Amit Sahai, Shai Halevi, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *In FOCS*, 2013.
9. James Hamilton and Sebastian Danicic. An evaluation of current java bytecode decompilers. SCAM '09, pages 129–136, Washington, DC, USA, 2009. IEEE Computer Society.
10. Ahmad Jbara and Dror G. Feitelson. On the effect of code regularity on comprehension. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 189–200, New York, NY, USA, 2014. ACM.
11. John C. Kieffer and En H. Yang. Sequential codes, lossless compression of individual sequences, and Kolmogorov complexity. *IEEE Trans. on Information Theory*, 42(1):29–39, 1996.
12. Ming Li and Paul M.B. Vitnyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Publishing Company, Incorporated, 3 edition, 2008.
13. Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.
14. Rabih Mohsen and Alexandre Miranda Pinto. Algorithmic information theory for obfuscation security. In *SECRYPT 2015 - Proceedings of the 12th International Conference on Security and Cryptography, Colmar, Alsace, France, 20-22 July, 2015.*, pages 76–87, 2015.
15. Mila Dalla Preda and Roberto Giacobazzi. Semantics-based code obfuscation by abstract interpretation. *Journal of Computer Security*, 17(6):855–908, 2009.
16. Jianhui Tian and Marvin V. Zelkowitz. A formal program complexity model and its application. *Journal of Systems and Software*, 17(3):253 – 266, 1992.
17. E. J. Weyuker. Evaluating software complexity measures. *IEEE Trans. Softw. Eng.*, 14(9):1357–1365, September 1988.