Imperial College London

Department of Computing

Quantitative Measures for Code Obfuscation Security

Rabih Mohsen

May 2016

Supervised by Steffen van Bakel

Submitted in part fulfilment of the requirements for the degree of

Doctor of Philosophy in Computing of Imperial College London

and the Diploma of Imperial College London

Declaration of Originality

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Rabih Mohsen

Copyright Declaration

Abstract

In this thesis we establish a quantitative framework to measure and study the security of code obfuscation, an effective software protection method that defends software against malicious reverse engineering. Despite the recent positive result by Garg et al.[GGH$^+$13] that shows the possibility of obfuscating using indistinguishability obfuscation definition, code obfuscation has two major challenges: firstly, the lack of theoretical foundation that is necessary to define and reason about code obfuscation security; secondly, it is an open problem whether there exists security metrics that measure and certify the current state-of-the-art of code obfuscation techniques. To address these challenges, we followed a research methodology that consists of the following main routes: a formal approach to build a theory that captures, defines and measures the security of code obfuscation, and an experimental approach that provides empirical evidence about the soundness and validity of the proposed theory and metrics. To this end, we propose*Algorithmic Information Theory*, known as Kolmogorov complexity, as a theoretical and practical model to define, study, and measure the security of code obfuscation.

We introduce the notion of unintelligibility, an intuitive way to define code obfuscation, and argue that it is not sufficient to capture the security of code obfuscation. We then present a more powerful security definition that is based on the algorithmic mutual information, and show that is able to effectively capture code obfuscation security. We apply our proposed definition to prove the possibility of obtaining security in code obfuscation under reasonable assumptions. We model adversaries with deobfuscation capabilities that explicitly realise the required properties for a successful deobfuscation attack.

We build a quantitative model that comprises a set of security metrics, which are derived from our proposed theory and based on lossless compression, aiming to measure the quality of code obfuscation security. We propose normalised information distance *NID* as a metric to measure code obfuscation resilience, and establish the relation between our security definition and the normalised

information distance. We show that if the security conditions for code obfuscations are satisfied (the extreme case) then the *NID* tends to be close to one, which is the maximum value that can be achieved.

Finally, we provide an experimental evaluation to provide empirical validation for the proposed metrics. Our results show that the proposed measures are positively correlated with the degree of obfuscation resilience to an attacker using decompilers, i.e. the percentage of the clear code that was not recovered by an attacker, which indicates a positive relationship with the obfuscation resilience factor.

Acknowledgements

Undertaking this PhD has been a truly life-changing experience and it would not have been possible to accomplish without the support and guidance that I received from numerous people.

I would like to express my special appreciation and thanks to my advisor Dr Steffen van Bakel, who taught me how to approach theory in computer science, to be precise and meticulous, also for his patience and continuous support of my PhD study.

I would also like to thank my committee members: Professor Chris Hankin and Dr David Clark, for letting my defence be an enjoyable challenge, and for their brilliant feedback, comments and suggestions.

Special thanks to Dr Emil Lupu for his insightful feedback, advice, and his unremitting support especially during the writing-up stage.

I have been very fortunate to meet Dr Alex Pinto, from whom I learnt immensely about Algorithmic Information Theory and writing academic papers, I also appreciate his precious collaboration.

I am extremely grateful to Dr Amani El-Kholy, for her kindness, patience, and assistance. I would like to express my profound gratitude to Professor Sophia Drossopoulou for her kindness, advice and unrelenting support. Furthermore, I am very grateful to Professor Marek Sergot; for his valuable guidance, patience and encouragement. His wise words helped me to fight my worries and concerns, and replaced them with great confidence enabling me to finish this thesis.

I would like to thank: Dr Herbert Wiklicky for increasing my interest in code obfuscation, Dr Maria Grazia Vigliotti and Dr Sergio Maffeis for the many interesting discussions on this work. I have also received support and help from many other academics, particularly Dr Krysia Broda and Professor Michael Huth especially during the first year of my PhD.

I would like to thank the Engineering and Physical Sciences Research Council, for providing the funding which allowed me to undertake this research.

Special thanks to my friends: Dr Makram Khoury-Machool for his sincere encouragement,

guidance and advice, Dr Shahram Mossayebi, Dr Dimitrios Kouzapas, and Ali I. Moussa, for being patient and supportive friends. They made this process bearable, especially during the hard and distressing times.

I would like to extend my sincerest thanks and appreciation to my parents, Zahra and Nazih, my sister, Abir, and my brothers: Ali, Mahdi, and Hussein, for their faith in me and for their altruism and emotional support. Finally, I wish to give my heartfelt thanks to my wife, Kamila, whose unconditional love, patience, and continual support enabled me to complete this thesis.

Contents

List of Figures

List of Tables

1 Introduction

The usage of software applications has become one of the corner stones of our lives. Nowadays, we are surrounded by software applications, like online payments applications, social networking, games, etc, and we have become more and more dependent on software and cyberspace even for our simple daily life tasks. Software often get distributed over the Internet; once distributed to a client machine, the software owner loses control over the software, especially as more applications and platforms become mobile. Many mobile wireless devices, including laptops, tablets, and smart phones, are becoming part of our daily lives. Developing professional and specialised software such as complex graphic processing applications and sensitive military software, require massive investment and efforts in terms of cost and money. Therefore, they can be very expensive and of great importance to their owners.

Attackers whether they are individuals, crime organisations or rogue regimes have great motivation and interest in stealing, reuse, tampering and reverse engineer such artefacts; often for the purpose of extracting secret information and/or proprietary algorithms. This type of attack is called a *malicious host-based* attack, which is conducted by malicious software or malicious users. Malicious host-based attacks operate in a *white-box model*; in this model, the attacker has full access to the system in which the software is running, and the attacker has a full privileged access to the system. This means that the malicious user can execute the program at her/his will, for example, s/he can observe the memory, processor, registers, and modify bytes during runtime.

Malicious reverse engineering is the most common type of malicious host-based attacks; it is used to reconstruct the program's source-code, and to conduct other host-based attacks, such as software piracy, reusing and tampering [CN09]. There are legitimate reasons for reverse engineering, for example software developers leverage reverse engineering to improve their own products, especially in the undocumented applications. In general, the aim of reverse engineering is to aid the process of understanding the inner working of software, and to obtain essential knowledge of the reversed

software in order to reuse it in other software. This process is unlawful if it violates the intellectual property rights of software owner such as extracting proprietary algorithms. Therefore, thwarting malicious reverse engineering is vital inhibiting host-based attacks.

There are different strategies to defend against host-based attacks, and to protect the working of software, such as both legal and technical countermeasures. Copyright and patent are two main approaches to protect software against unlawful copying and stealing of algorithms. Despite the fact that copyright protection defends against illegal copying, it does not help in protecting the idea or the implemented algorithms. Software patents help to protect the computer programs inventions including the idea and the algorithm; however, they do not provide solid protection as they are not always enforceable. The major drawback of such patents is the cost. There are usually very expensive to enforce, and therefore unaffordable for small companies. According to the US Digital Millennium Copyright Act (DMCA) and EU Computer Programs Directive legislations, reverse engineering is allowed for the purpose of interoperability between computer programs, if the programs are obtained lawfully. Hence, it is very difficult under these regulations to prevent reverse engineering for the understanding of the inner working of software. Because of these shortcomings, legal protection mechanisms, in most cases, have a small impact on foiling malicious host-based attacks.

Technical measures present a cheaper alternative to protect against malicious host attacks. Technical protection increases the cost of extracting secret information and reverse engineering for malicious purposes—even on the current open computing platforms, where software's execution is relatively easy to inspect and modify. This process is conducted by following one of two paths: hardware and software protection. Hardware protection techniques leverage the hardware devices capabilities to provide protection, such as secure coprocessors, Trusted Computing (TC), tamper resistance, and smart cards, where secure computation is carried inside the protected hardware despite deployment in a hostile computing environment. However, hardware protection techniques do not provide a complete solution to the malicious host-based attacks, and their logistic challenges (such as the difficulty of upgrading hardware) usually create difficulties in adapting them to computing infrastructures. For example, if the hardware protection technique gets compromised by an attacker, it would be very difficult to provide a quick response to patch and fix this problem. It would require a full upgrade and replacement cycle to get the device secure again [Ird13]. Moreover, hardware protection devices have a high cost and suffer from compatibility issues with other open

computer platforms.

Software protection techniques, on the other hand, provide security by preventing inspections, modifications, and reverse engineering. Software defence approaches are more flexible as they are less platform dependent, and cheaper, than their hardware counterpart. There are different forms of software protection, for example remote executions, encryption and authentication. Remote execution, such as server-side execution, treats software as a service, so that the software is not distributed but run as a service. These techniques stop the hostile host from having physical access to the secure software by running it remotely, in a similar way to *black-box model* as the attacker can only analyse the external behaviour of the software (input-output). However, they require reliable network communication and are prone to performance degradation.

Encryption and authentication methods help to secure software and sensitive data, and decrypt the software code on the fly during the execution process. Unfortunately, since the execution requires decryption, the clear (decrypted) code is revealed in the memory during execution, and the attacker can dump the memory and construct the code. Therefore, encryption and decryption have to be conducted by trusted hardware devices, and thus it suffers from the same hardware protection drawbacks.

Software-tamper resistance techniques help to shield software against modifications and tampering by creating checksums and hash-codes for detection in the protected code. Beside detection, they take actions post the detection phase, which may include disabling, deleting, or making the software generate invalid results, rendering it useless to the reverse engineering attacks.

If the tamper resistance process is circumvented, then the software is left without defence against malicious host-based attacks. One of the defence methodologies that can be effectively used in this case, which is the subject of this thesis, is *code obfuscation*. In essence, the purpose of code obfuscation is to make the code difficult to read and understand, and hard to analyse by attackers, while preserving the intended functionality of the original program. The basic premise being that if the attacker cannot understand the outcome of the reverse engineering, then it is virtually impossible to usefully alter the reversed engineered code.

1.1 Code Obfuscation

Obfuscators attempt to manipulate code in such a way that it becomes unintelligible to human, and automated program analysis tools that are used by malicious reverse engineers, while preserving the functionality. Code obfuscation is a low-cost technique that does not cause a concern for portability and is promising for defending mobile programs against malicious host-based attacks [CN09].

In general, code obfuscation helps shield the inner working of the code against analysis attacks. We enumerate some cases of software protection scenarios, where code obfuscation can be useful at making attacks economically infeasible.

Protection of intellectual property. Decompilation and reverse engineering allow attackers to understand the code, extracting proprietary algorithms and data structures (e.g. cryptographic keys) from software. Code obfuscation defends against malicious decompilation and reverse engineering, by complicating the analysis phase in this process. Thus, this process should become very costly and too expensive, in terms of the required time or resources.

Code lifting. In this type of attack, the adversary tries to identify the sections in the code of interest, rather than understanding the overall code. Code obfuscation does not stop copying the code or part of it; however it helps to make the process of identifying these parts more difficult.

Hiding vulnerabilities. This idea is based on the assumption that making the process of analysis and understanding of the inner workings of a program harder, using code obfuscation, helps to prevent attackers from discovering vulnerabilities in the code.

Software watermarking. Software watermarking is used to prove the ownership of the software, normally through embedding a unique identifier within software that binds the program to a specific user. Software watermarking does not stop software privacy, however it discourages software theft, by proving to authorities the ownership of stolen software. The typical attack on software watermarking is to identify the watermark and destroy it; code obfuscation can be one of the protection techniques to prevent this attack, in a similar way to code lifting, by making it harder to recognise the watermark.

Software birthmark. One of the software protection areas that are related to code obfuscation is software birthmark [CN09]. A software birthmark is one or a series of unique and inherent

characteristics that can be used to identify software theft, detect software piracy and identify malware such as viruses and trojan horses. Code obfuscation applies many changes to the code, which render detecting these birthmarks difficult.

Transform Private-Key Encryption to Public-Key Encryption. Obfuscation, when established in a well-defined security, i.e. the possibility of having proven theoretical foundations for security [BGI+01], could solve many problems which have not been addressed fully by cryptography. Transforming a Private-Key Encryption (Symmetric Encryption) into Public-Key Encryption is an example where Obfuscation can be used to transform Private-Key encryption schemes by obfuscating a symmetric (private)-key encryption scheme. Given a secret key k of a symmetric-key encryption scheme, one can publish an obfuscation of the encryption algorithm Enc_k. Hence, everyone can encrypt, but only the one who possesses the secret key k should be able to decrypt.

These are some legitimate usages of code obfuscation. However, code obfuscation is a double-edge sword; it can be used also for malicious purposes, for example in malware design. Code obfuscation[1] is used in malware for the purpose of evading anti-malware tools. Most virus scanners (AV) are signature based, and malware make these tools fail by using code obfuscation. Obfuscation changes the structure and syntax of the malware while it preserves its behaviour, which makes tracing, dissembling and detection analysis more difficult to perform.

1.2 Motivation and Objectives

As discussed above, code obfuscation provides a promising technical approach for protecting software. However, most of the current state-of-the-art obfuscation techniques are not based on well-defined security principles that help to certify their success in protecting software. In essence, there are two related challenges that arise in code obfuscation security: the lack of a rigorous theoretical foundation, and the difficulty of finding consistent and, theoretically and empirically, valid measures of code obfuscation quality.

[1] Using different terminologies such as metamorphism and polymorphism.

Lack of a theoretical background for code obfuscation

A major challenge in the field of code obfuscation is the lack of a rigorous theoretical basis. The absence of a theoretical basis makes it difficult to formally analyse and certify the effectiveness of these techniques in safeguarding against malicious host-based attacks [PG09]. In particular, it is hard to compare different obfuscation transformations with respect to their resilience to attacks.

The impossibility result of finding a generic code obfuscation by Barak et al. [BGI+01] demonstrates the theoretical impediment to establishing a robust framework with well-defined security requirements, as in cryptography, for code obfuscation. Barak et al. [BGI+01] provide a formal definition of *perfect* obfuscation in an attempt to achieve well-defined security that is based on the black-box model. Intuitively, a program obfuscator $\mathcal{O}$ is called perfect if it transforms any program P into a 'virtual black-box' $\mathcal{O}(P)$ in the sense that anything that can be efficiently computed from $\mathcal{O}(P)$, can be efficiently computed given just oracle access to P. They proved that the black-box definition cannot be met by showing the existence of a set of functions that are impossible to obfuscate.

On the other hand, a recent study by Garg et al. [GGH+13] has provided positive results, using indistinguishability obfuscation, for which there are no known impossibility results. Two programs P and Q of the same size that compute the same functionality are computationally indistinguishable if no polynomial-time adversary can distinguish between the obfuscation of $\mathcal{O}(P)$ and the obfuscation of $\mathcal{O}(Q)$. However, as argued by [GR07] there is a disadvantage in indistinguishability obfuscation: it does not give an intuitive guarantee about the security of code obfuscation. Furthermore, indistinguishability obfuscation does not certify or reason about the current state-of-the-art obfuscation techniques (practical obfuscation techniques).

The obfuscation definition that was proposed by Barak et al. is based on the virtual black-box model, which is a very strong model for software protection, is an unrealistic expectation of obfuscation for two main reasons. First, software run on an open computing platform, and for this reason, any malicious host-based attacker has unlimited access to execute and modify the protected code. Secondly, we cannot expect a defender to hide program's functionality,[2] as virtual black-box model does for code obfuscation. Therefore, comparing any adversary who has access to an obfuscated program based on the virtual black-box model is impractical, and hence, led to the

[2]People are not interested in buying software that they do not known what it does (the programs functionality).

impossibility results.

The recent advances in code obfuscation theory by Garg et al. confirmed our argument; the indistinguishability obfuscation definition showed the possibility to obfuscate securely. The success behind indistinguishability obfuscation is based on eliminating the virtual black-box from the definition of obfuscation. Nevertheless, this theoretical success has not yet conceived any practical obfuscation algorithms. Consequently, the current state-of-the-art obfuscation techniques are predominantly the only available techniques for code obfuscation; this necessitates a new direction of research which allows to reason effectively over the security aspects of practical obfuscation techniques.

To this end, we need a new formal approach and framework for modelling, designing, studying and relating obfuscating transformations. Furthermore, a systematic methodology for deriving program transformations is highly desirable in order to design obfuscating algorithms, which are able to hide a desired property, and to defeat a given attacker. Toward achieving these goals, it is crucial to provide new definitions for practical code obfuscation, in particular, and software protection in general.

The problem of finding code obfuscation metrics

The current notion of code obfuscation is based on a fixed metric for program complexity, which is usually defined in terms of syntactic program features, such as code length, number of nesting levels and numbers of branching instructions. There is a need to practically examine and verify the effectiveness of obfuscation transformation based on new quantitative means [PG09].

Most code obfuscation strategies are ad hoc and their metrics rely on software metrics that are based on classical complexity metrics. Despite the usefulness of such metrics, they fall short of achieving, quantitatively, the confidence and security trust level in code obfuscation.

Software similarity metrics could provide a way to reason about the amount of confusion added by obfuscation transformation techniques. However, we believe, in their current state, they are not adequate to measure the quality of code obfuscation. Two programs P and $\mathcal{O}(P)$ can be similar or dissimilar with a certain value v, but, that does not give any information on the code obfuscation security, i.e. how difficult it is to comprehend the protected properties in the transformed program $\mathcal{O}(P)$. In order to evaluate the quality of software protection such as code obfuscation, we have to capture quantitatively the security of code transformations, and study the code-obfuscation

resilience against an adversary, taking into account the adversary's capabilities, such as malicious software reverse engineering, static and dynamic analysis techniques, etc.

1.3 Contributions

In an attempt to tackle these problems, we followed two research methodologies. The first formal approach is used to build a theory that defines and reasons about code obfuscation security; the approach is based on *Algorithmic Information Theory*, which is also known as *Kolmogorov complexity* [Kol65]. The Kolmogorov complexity for a binary string s is defined as the length of the shortest program that runs on a universal Turing machine and produces s. Kolmogorov complexity measures the intrinsic information content and randomness in a given string. Kolmogorov complexity is uncomputable; however, it can be approximated using lossless data compression [KY96]. In this thesis Kolmogorov complexity serves as a unified notion to define and provide proofs for code obfuscation security, and to derive a theoretical and practical model that establishes a set of quantitative metrics, which is used to measure the quality of code obfuscation.

The second approach is based on experimental evaluation; we provide empirical evidence to assess and validate the soundness and effectiveness of the derived metrics measuring the security of code obfuscation. We correlate the value of these metrics with a heuristic-based obfuscation resilience factor using decompilation attacks (percentage of failing to retrieve the original clear code).

Based on the outcome of applying these research methodologies, we make the following contributions.

A new theoretic framework for practical code obfuscation security. We provide a theoretical framework for code obfuscation in the context of Algorithmic Information Theory (Kolmogorov complexity), to quantitatively capture the security of code obfuscation. Our definition allows for a small amount of secret information to be revealed to an adversary, and gives an intuitive guarantee about the security conditions that have to be met for practical secure obfuscation. We argue that our model of security is fundamentally different from the virtual black-box model of Barak et al. in which their impossibility result does not apply. We assume the functionality of an obfuscated program to be almost completely known and available to an adversary, and only require hiding the implementation rather than the

functionality itself. This approach to obfuscation is very practical and pragmatic, especially for software protection obfuscation. We show that our model is very similar to indistinguishability obfuscation, yet it has an advantage over the obfuscation indistinguishability definition in the sense it is more intuitive, and is algorithmic rather than probabilistic. We then show that under reasonable conditions we can have secure obfuscation. We investigate the security of the two main approaches to obfuscated code in software, *encoding* and *hiding*, at the subprogram level. Moreover, we study the effect of combining several obfuscation techniques in the same program, and investigate their overall security.

Modelling deobfuscation adversaries. We model deobfuscation adversaries using Algorithmic Information Theory, and define the security properties that characterise the conditions for successful code obfuscation attack.

Metric theoretical evaluation for Kolmogorov complexity. We apply the Weyuker validation framework [Wey88] to check whether Kolmogorov complexity is theoretically sound as a software metric. The results show that Kolmogorov complexity is a suitable metric for measuring complexity in binary programs, and code obfuscation in particular.

Quantitative metrics to measure the quality of code obfuscation. We propose a new quantitative framework to measure the quality of code obfuscation; we rely on lossless data-compression algorithms to approximate Kolmogorov complexity, and to have a practical means to measure the regularity (randomness) in code obfuscation. We show that software similarity metrics such as *information distance* [LCL+04] that measures the similarity between two blocks of code, can provide a plausible way to reason about the amount of security added by code obfuscation transformation. The aim of using information distance is to quantify the amount of obscured code that remains or is lost when the program is debofuscated. We formalise the notions of *unintelligibility index* (degree of confusion introduced) and *relative Kolmogorov complexity*, and show that information distance metric is a suitable measure for code obfuscation resilience. We also apply a modified version of information distance to define code obfuscation stealth, and propose a statistic model based on linear regression that combines all the proposed metrics to estimate the total security of code obfuscation.

Empirical evaluation for the proposed metrics. The empirical validation results show that the

proposed metric are empirically valid for measuring the quality of code obfuscation. These metrics outperforms the classical complexity measures in terms of being correlated with the degree of code obfuscation's resilience to decompilers. Moreover, the outcome of the analysis of the results shed a light on the importance of taking into account the attack model when measuring the quality of code obfuscation. Applying any quantitative measure without parametrising it to a specific attacker can be misleading, in that it creates a false sense of security.

1.4 Related Work

The first attempt to evaluate obfuscation was conducted by Collberg et al. [CTL97]; they relied on classical software complexity metrics to evaluate obfuscation such as Cyclomatic Complexity, and Nesting Complexity. Anckaert et al. [AMDS+07] suggested a framework of four program properties that reflect the quality of code obfuscation: code, control flow, data and data flow. They applied software complexity metrics to measure these properties; however they did not perform any validation on the proposed metrics. Ceccato et al. [CPN+09] 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. [CCFB14], a set of software metrics (modularity, size and complexity of code) were applied to a set of obfuscated programs to measure the complexity and potency of obfuscated programs. Their results showed that a limited number of obfuscated techniques, involved in their study, were effective in making code metrics change substantially from original to obfuscated code. We apply a similar statistical validation methodology to evaluate the proposed metrics, yet based it on non-parametric statistical techniques and regression analysis [She07]. However, our approach differs substantially from their approach; they applied classical complexity metrics, where we apply new quantitative metrics using Algorithmic Information Theory and compression.

The most related work to our approach is the work that was conducted by Kirk et al.[KJ04] who investigated the possibility of using information theory to measure code obfuscation through plain Kolmogorov complexity and compression to measure the level of randomness in code obfuscation.

However, their work lacks the theoretical and empirical evaluations that we provide in this thesis.

Jbara et al. [JF14] argued that most of the complexity metrics are syntactic program's features that ignore the program's global structure. The global structure of a program may have an effect on the understanding of that program, and they suggested the use of code regularity, which is estimated by compression, to measure program comprehension, and 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 [KY96]. Their intuitions and results agree with our observation and theoretical treatments 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. They only used compression to measure code comprehension in an empirical sense, without applying any theoretical validation. Furthermore, they did not apply their experiment to study the effect of compression on obfuscated code.

1.5 Thesis Layout

This thesis is organised in the following chapters:

Chapter 2. We provide the mathematical preliminaries that we are going to use in this thesis, together with a brief introduction to computability theory, Turing machines, Information Theory and Algorithmic Information Theory (Kolmogorov complexity).

Chapter 3. We present the current threats to software such as program analysis and malicious reverse engineering, and present the untrusted host as an attack model for software. We introduce the notion of code obfuscation as a potential defence method against such an attack. We also provide some of the theoretical background in this domain, in particular, the impossibility results of virtual black-box obfuscation, and the latest advances that are based on a relaxed version of virtual black-box model, which provide positive results on code obfuscation. We provide an overview of obfuscating techniques based on Collberg et al.'s [CTL97] taxonomy on code obfuscation algorithms. We discuss some of deobfuscation

methodologies that can be used to foil code obfuscation techniques. Finally, we present an overview of metrics, which are currently employed to measure the quality of code obfuscation.

Chapter 4. We motivate the use of Algorithmic Information Theory in code obfuscation. We introduce the notion of unintelligibility to define confusion in code obfuscation and argue that this is not good enough. We then propose our notion of security that is based on algorithmic mutual information, and compare both definitions, in particular, with the virtual black-box model and indistinguisibility obfuscation. Then we apply our security definition to study the security of the two main approaches to obfuscated code in software, *encoding* and *hiding*, at the sub-program level. We also investigate the effect of combining multiple obfuscation techniques and reason about their security.

Chapter 5. We propose a generic model of a code obfuscation adversary based on Algorithmic Information Theory and Kolmogorov complexity. We present a formal grounding and a new definition of a code obfuscation adversary that captures the adversary's objectives and capabilities.

Chapter 6. In this chapter we propose a model to measure code obfuscation quality that is based on our theoretical investigation in Chapter 4 and Chapter 5. The model comprises of four different metrics: unintelligibility index, normalised Kolmogorov complexity, normalised information distance for resilience and code obfuscation stealth. We check whether Kolmogorov complexity is theoretically sound as a valid software metric based on Weyuker's validation framework [Wey88]. We show that information distance [LCL$^+$04] can provide a plausible way to reason about the amount of security added by code obfuscation transformation.

Chapter 7 We provide the experimental design and tool-sets that are necessary to conduct and interpret the results of evaluating the metrics that were proposed in Chapter 6. The experiment consists of a set of obfuscated Java jar files of SPECjvm2008 benchmark, using two obfuscators: Sandmark, an open source suite, and Dasho, a commercial tool, and three different decompilers as an attack model.

Chapter 8. In this chapter we focus on the experimental results and analysis. We present the research questions, the formulated null hypotheses, and show the validation results of the

proposed metrics model, using statistical hypotheses testing, in addition to other statistical tools. We also provide comparison with the classical complexity metrics. Finally, we examine the impact of code obfuscation using the proposed metrics.

Chapter 9: We sum up the main contributions of this thesis and briefly describe the directions for future work.

1.6 Publications

1. 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.

2. Rabih Mohsen and Alexandre Miranda Pinto. Evaluating obfuscation security: A quantitative approach. In *Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers* , pages 174–192, 2015.

3. Rabih Mohsen and Alexandre Miranda Pinto. Theoretical Foundation for Code Obfuscation Security: A Kolmogorov Complexity Approach. *E-Business and Telecommunications: 12th International Joint Conference, ICETE 2015, Colmar, France, July 20–22, 2015, Revised Selected Papers*, pages 245–269. Springer International Publishing, Cham, 2016.

2 Preliminaries

The goal of this chapter is to give a reasonably self-contained account of the basics, as well as some of the mathematical tools we will need in this thesis.

2.1 Notations

We use the notation U as shorthand for a universal Turing machine, x for a finite-length binary string and $|x|$ for its length. For a set S, the count of its elements are denoted by $\#S$. $\bar{s}$ is used to denote the complement of $s \subseteq S$ such that $\bar{s} = S \backslash s$. We use ϵ for a negligible value, $p(n)$ for a polynomial function with input $n \in \mathbb{N}$, and $O(1)$ for a constant. $\|$ is used to denote the concatenation between two programs or strings.

In this thesis, we are only using *strings* over a set of binaries i.e. $\{0,1\}$. The set of all finite strings over $\{0,1\}$ is denoted by $\{0,1\}^*$, which we use to represent the space of all possible strings including the empty string, and can be formally defined as

$$\{0,1\}^* = \bigcup_{i=0}^{\infty} \{0,1\}^i$$

where $\{0,1\}^0 = \varepsilon$ is an empty string, and $\{0,1\}^{n+1} = \{0S \mid S \in \{0,1\}^n\} \cup \{1S \mid S \in \{0,1\}^n\}$.[1] All objects, such as natural numbers and program code, are encoded as binary strings. For a given string $x \in \{0,1\}$, $|x|$ is measured in the number of symbols of that string. We also write $\{0,1\}^+ = \{0,1\}^* \backslash \varepsilon$ to denote the space of all possible strings excluding the empty string.

Following this, we may inductively create a rule that allows us to totally order all strings that are possible in $\{0,1\}^*$ in a conventional way, according to their length. Then we associate each string by a natural number [2], this number act as an index in the length-increasing lexicographic ordering.

[1] The symbol $*$ refer to a closure of the set, i.e. $\{0,1\}^*$ is closed under the operation of concatenation; that is, if x and y are belongs to $\{0,1\}^*$, then $x \| y$ are in $\{0,1\}^*$ too.

[2] This scheme will make the string easily decodable.

In this case, every string in $\{0,1\}^*$, can be identified by its index in the ordering [LV08].

$$(\varepsilon,0),(0,1),(1,2),(00,3),(01,4),(10,5),(11,6),....$$

The length of a string x, $|x|$ is related to the index or position of x in the above relation, which can be computed using logarithmic term such that: $|x| = \lceil \log(x+1) \rceil$, where $\lceil . \rceil$ denotes the ceiling operation that returns the smallest integer not smaller than the argument [LV08].

$\mathcal{P}$ is a set of binary programs and $\mathcal{Q}$ is a set of binary obfuscated programs, $\mathcal{L} = \{\lambda_n \mid \lambda_n \in \{0,1\}^+, n \in \mathbb{N}\}$ is a set of (secret) security parameters that is used in the obfuscation process.[3] $\mathbb{A} = \{\mathcal{A}_n \mid n \in \mathbb{N}\}$ represents a set of adversaries (deobfuscators) where an adversary $\mathcal{A} \in \mathbb{A}$ uses a set of deobfuscation techniques (e.g. reverse engineering); the term adversary is used interchangeably with deobfuscator. We say two binary programs P and Q have the same functionality (meaning) if they produce the same output given an input and terminate, i.e. given an input set I, $[\![P]\!] = [\![Q]\!] \iff \forall i \in I.[P(i) = Q(i)]$.

2.2 Probability

Probability theory deals with predicting how likely it is that something will happen. For example, if one tosses three coins, how likely is it that all will come up heads? The notion of the likelihood of something is formalised through the concept of an experiment (or trial) - the process by which an observation is made. In this technical sense, tossing three coins is an experiment.

Definition 2.1. *The set of all possible experimental outcomes is called the* sample space *and is denoted by* Ω.

Sample spaces may either be discrete, having at most a countably infinite number of basic outcomes, or continuous, having an uncountable number of basic outcomes.

The foundations of probability theory depend on the set of events $\mathcal{F}$ forming a σ-field, a set with a maximal element Ω and arbitrary complements and unions. These requirements are trivially satisfied by making the set of events, the event space, the power set of the sample space.

Definition 2.2. *A set* $\mathcal{F} \subseteq \Omega$ *is called a* σ*-field if:*

[3] The security parameter may include the obfuscation key, the obfuscation transformation algorithm or any necessary information that the obfuscation function can use.

1. $\varnothing \in \mathcal{F}$

2. *if* $A_1, A_2, ... \in \mathcal{F}$ *then* $\bigcup_{i=1}^{\infty} A_i \in \mathcal{F}$

3. *If* $A \in \mathcal{F}$ *then* $\bar{A} \in \mathcal{F}$

Definition 2.3 ([Sti05])**.** *A probability measure* $\Pr$ *on* $(\Omega, \mathcal{F})$ *is a function* $\Pr : \mathcal{F} \to [0, 1]$ *satisfying*

1. $\Pr(\varnothing) = 0$,

2. $\Pr(\Omega) = 1$,

3. *If* $A_1, A_2, ...A_n$ *is a collection of pairwise disjoint members of* $\mathcal{F}$*, i.e. if* $A_i \cap A_j = \varnothing$ *for* $j \neq j$ *where* $A_i, A_j \in \mathcal{F}$*, then*

$$\Pr(\bigcup_{i=1}^{n} A_i) = \sum_{i=1}^{n} \Pr(A_i)$$

$\Pr$ *is also called the* probability distribution*. The entire structure comprising* Ω*, with its event space* $\mathcal{F}$*, and probability function* $\Pr$*, is called the* probability space *and is denoted by (*Ω*,* $\mathcal{F}$*,* $\Pr$*).*

Lemma 2.4 (Basic Properties)**.** *For any events* $A, B \subseteq \Omega$*, we have the basic properties:*

1. $\Pr(\bar{A}) = 1 - \Pr(A)$*, where* $\bar{A} = \Omega \backslash A$,

2. $\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B)$,

3. $\Pr(A) \leq \Pr(B)$*, if* $A \subseteq B$,

4. $\Pr(A \cup B) = \Pr(A) + \Pr(B)$ *if A and B are disjoint.*

Conditional Probability and Independence

Sometimes we have partial knowledge about the outcome of an experiment, which naturally influences what other experimental outcomes are possible. We capture this knowledge through the notion of conditional probability. The probability of an event before we consider our additional knowledge is called the prior probability of the event, while the new probability that results from using our additional knowledge is referred to as the posterior probability of the event [LG07].

Definition 2.5 (Conditional Probability)**.** *The conditional probability of an event A given that an event B occurs, if $\Pr(B) > 0$, is defined by*

$$\Pr(A \mid B) = \frac{\Pr(A \cap B)}{\Pr(B)}$$

In general, the occurrence of some event B changes the probability that another event A occurs, the original probability $\Pr(A)$ will be replaced by $\Pr(A \mid B)$. If the probability remains unchanged, then $\Pr(A \mid B) = \Pr(A)$ and A, B are called independent.

Definition 2.6. *Events A and B are called independent if*

$$\Pr(A \cap B) = \Pr(A) \Pr(B)$$

More generally, a family of events $\{A_i \mid i \in I\}$ is called independent if

$$\Pr(\bigcap_{i \in J} A_i) = \prod_{i \in J} \Pr(A_i)$$

for all finite subsets J of I.

2.2.1 Random Variable

Rather than having to work with some irregular event space which differs with every problem, we look at a random variable that allows us to talk about the probabilities of numerical values that are related to the event space, without having to exhibit all the events in $\mathcal{F}$.

Definition 2.7 (Random Variable [Sti05])**.** *A random variable is a function $X : \Omega \to \mathbb{R}$ with the property that $\{\omega \in \Omega \mid X(\omega) \leq x\} \in \mathcal{F}$ for each $x \in \mathbb{R}$.*

Every random variable has a distribution function, which is the probability that the random variable X does not exceed x.

Definition 2.8 (Discrete Random variable [Sti05])**.** *The random variable X is called discrete if it is taking some countable subset $x_1, x_2, \ldots$ of $\mathbb{R}$. The discrete random variable X has probability function $f : \mathbb{R} \to [0, 1]$ given by $f(x) = \Pr(X = x)$.*

Definition 2.9. *The* expected value E *of a discrete random variable* X *taking values* $x_1, ..., x_n$ *is:*

$$E[X] = \sum_{i=1}^{n} x_i \Pr(x_i)$$

2.3 Information theory

The field of information theory was developed in the 1940s by Claude Shannon, with the initial exposition reported in [Sha48]. Shannon was interested in the problem of maximizing the amount of information that you can transmit over an imperfect communication channel such as a noisy phone line (though actually many of his concerns stemmed from codebreaking in World War II). For any source of 'information' and any 'communication channel' Shannon wanted to be able to determine theoretical maxima for (i) data compression, which turns out to be given by the Entropy H (or more fundamentally, by the Kolmogorov complexity K (see Section 2.7)), and (ii) the transmission rate, which is given by the Channel Capacity [CT06]. Before 1948, people had assumed that necessarily, if you send a message at a higher speed, then more errors must occur during the transmission. But Shannon showed that provided that you transmit the information in the message at a slower rate than the Channel Capacity, you can make the probability of errors in the transmission of your message as small as you would like.

The initial questions treated by information theory lay in the area of data compression and transmission. The answers are quantities such as entropy, mutual information, and relative entropy [CT06], which are functions of the probability distributions that underlie the process of communication.

2.3.1 Entropy

Given a discrete random variable X we cannot know for sure which of its values $\{x_1, x_2, ...\}$ will occur. Shannon introduced the concept of entropy which is a measure of the uncertainty of a discrete random variable [CT06].[4] It is the number of bits on average that are required to describe the random variable.

Definition 2.10 (Entropy [CT06])**.** *Let* X *be a discrete random variable over symbols (alphabet)* $X = \{x_1, ..., x_n\}$ *and probability function* $\Pr(x_i) = \Pr(X = x_i), x_i \in X$*. The entropy* $H(X)$ *of*

[4] Entropy can be also defined for continuous random variables.

the discrete random variable X is defined by :

$$H(X) = -\sum_{i=1}^{n} \Pr(x_i) \log \Pr(x_i).$$

The $\log$ is to the base 2 and it is measured in bits. Note that entropy is a function of the distribution of X; it does not depend on the actual values taken by the random variable X, but only on the probabilities.

2.4 Prefix Codes

A string x is called a *proper-prefix* of another string y, if for a string $z \neq \varepsilon$, $x = y \parallel z$. A set $S \subset \{0,1\}^*$ is *prefix-free* if no element is a proper-prefix of any other. A prefix-free set is used to define a *prefix-code*. A function $D : \{0,1\}^* \to \mathbb{N}$ defines a prefix-code if its domain, $\{0,1\}^*$, is prefix-free; this function is called a *decoding* function.

A binary string y is a *code-word* (words of the code alphabet) for *source-word* (words from the source alphabet) x if $D(x) = y$, and D is the decoding function. The set of all code-words for a source-word x is the set $E = D^{-1}(x) = \{y \mid D(y) = x\}$, and is called the encoding function [LV08].

The interest in prefix-codes is motivated by the need for uniquely decodable codes. If no code-word is the prefix of another code word, then each code sequence is uniquely decodable, since the set of source words is infinite ($\mathbb{N}$) and we would have to use variable-length codes.

Definition 2.11 (Prefix-code [CT06])**.** *A code is a* prefix-code *or* instantaneous code *if the set of code-words is prefix-free.*

An example of prefix-free encoding for numbers, is to use a self-delimiting code; for a binary string x such as [5]

$$\hat{x} = 1^{|x|}0x$$

In this case, the encoding function $E(x) = \hat{x} = 1^{|x|}0x$, where 0 is the stop symbol. This is a type of prefix-free code that is called a self-delimiting, since there is a fixed computer program linked to this code, which determines where the code-word $\hat{x}$ ends by reading it from left to right without

[5] Here $x \in \mathbb{N}$ which is represented by binaries so that $x \in \{0,1\}^*$.

backing up. Using this encoding, we can construct a prefix-free set such as [LV08]

$$\{\hat{x} \mid x \in \{0,1\}^*\}$$

Applying this method, the code can be parsed in its constituent code-words in one go by a computer program. It is desirable to construct instantaneous codes of minimum expected length to describe a given source; this is very crucial in data compression because we need to have code-words of shortest possible length for encoding and decoding. It is obvious that we cannot assign short code-words to all source symbols, for the purpose of unique encoding-decoding, and still be prefix-free. The set of code-word lengths that are possible for instantaneous codes is restricted by the following inequality, known as the Kraft inequality.

Theorem 2.12 (Kraft Inequality [Kra49])**.** *Let $l_1, l_2, .., l_n$ be the code-word lengths for each of n code-words in a binary prefix-free code. Then,*

$$\sum_{i=1}^{n} 2^{-l_i} \leq 1$$

2.5 Computability Theory

The notion of computability is a basic principle in computer science, as it defines what a computer can do. The classical computability theory originated with the seminal work of Gödel, Church, Turing, Kleene and Post in the 1930's [Rob15]. Intuitively, computation is a process that produces some output on certain input using a specific set of rules, which dictates how to perform the computational operation. Computability theory is also known as *recursion theory*; it is concerned with studying whether functions are computable or not. In essence, the classical computability theory is the theory of functions on the integers that are computable by a finite procedure. The computable functions are the fundamental object of study in computability theory. This includes computability on many countable structures as they can be coded also by integers.

Models of computation are abstract specifications of how a computation can be performed, which are expressed as the description of some kind of conceptual automaton. We need a model of computation in order to abstract from implementation details using any programming language. The most popular mathematical model for computability and computations is due to Alan Turing,

who defined what is now known as Turing machines.

Different approaches to computability were introduced, using models that are based on recursive functions, the first one by Gödel [Rob15] then advanced by Kleene [Kle64] and Church [Chu32] using *Lambda-Calculus*, which expresses computation based on function abstraction. All the proposed models of computations are equivalent in the sense that they express the same class of computable functions. Church and Turing provided two equivalent theses for computable functions, which later were merged into a well known *Church-Turing Thesis*. It states that any partial function that is computable in any model is also computable by a *Turing Machine*[6]: these functions are called *Turing-computable*. The Church-Turing Thesis is a statement that is believed to be true but is not proven, based on the fact that many computation models are equivalent, and so far no one has presented a formal proof to reject this statement [Rob15]. In the following, we give a formal definition of recursive functions according to Gödel's model.

Definition 2.13 (Primitive Recursive Functions [vB15])**.** *The class of* primitive recursive *functions* $\mathcal{F}_R r$ *in* $\mathbb{N}^k \to \mathbb{N}$*, for any* k*, is constructed by:*

1. *The* initial *functions*

$$Z(x) = 0 \qquad \textit{(Zero)}$$

$$S(x) = x + 1 \qquad \textit{(Successor)}$$

$$P_n(x_1, ..., x_n) = x_i \ (1 \leq i \leq n) \qquad \textit{(Projection)}$$

2. Composition. *If* $g_1, ..., g_m : \mathbb{N}^k \to \mathbb{N}$*,* $h : \mathbb{N}^m \to \mathbb{N} \in \mathcal{F}_r$*, then* $f : \mathbb{N}^k \to \mathbb{N}$ *defined by*

$$f(x_1, ..., x_k) = h(g_1(x_1, ..., x_k), ..., g_m(x_1, ..., x_k))$$

   is in $\mathcal{F}_r$

3. Recursion. *If* $g : \mathbb{N}^k \to \mathbb{N}$*,* $h : \mathbb{N}^{k+2} \to \mathbb{N} \in \mathcal{F}_r$ *then* $f : \mathbb{N}^{k+1} \to \mathbb{N}$ *defined by*

$$f(0, x_1, ..., x_k) = g(x_1, ..., x_k)$$

$$f(y + 1, x_1, ..., x_k) = h(y, f(y, x_1, ..., x_k), x_1, ..., x_k)$$

[6]The Turning machine term can be substituted with any equivalent model of computation.

is in $\mathcal{F}_r$

Using the above definition, we can define *partial recursive* functions.

Definition 2.14 (Partial Recursive Function [vB15])**.** *The class of* partial recursive *functions* $\mathcal{F}_{pr} \in \mathbb{N}^k \to \mathbb{N}$ *(for any* k*) is defined as the set of primitive recursive functions with additional condition:*

4. *μ-Operation. If* $f : \mathbb{N}^{n+1} \to \mathbb{N} \in \mathcal{F}_{pr}$*, then* $h : \mathbb{N}^n \to \mathbb{N} \in \mathcal{F}_{pr}$*, where* h *is defined by:*

$$h(x_1, ..., x_n) = \mu y(f(y, x_1, ..., x_n)) = 0$$

where μy *expresses the least* $y \in \mathbb{N}$*, which causes* $f(y, x_1, ..., x_n)$ *to return 0.*

That means y begin with 0 and goes upward, until y is found, i.e. stepping through $f(0, x_1, ..., x_n), f(1, x_1, ..., x_n), f(2, x_1, ..., x_n), ...$
However, it might be the case that such argument does not exist, then the search might not terminate, and therefore, h is not defined.

Definition 2.15 (Total Recursive Function)**.** *A function is called (total) recursive if it is partial recursive and total, i.e. its domain is all of* $\mathbb{N}^k$*.*

Sets that can be algorithmically generated are called *recursive (computably) enumerable*.

Definition 2.16 (Recursive Enumerable Set [LV08])**.** *A set* S *is recursively enumerable if it is empty or in the range of some total recursive function* f*, so* f *effectively enumerates* S*.*

The intuition that motivates this definition is that there exists a Turing machine which lists the elements of S. For every element in S, the Turing machine halts in a distinguished accepting state, and the elements that are not in S the machine either halts in a non-accepting state or computes forever [LV08].

2.5.1 Turing Machines

A Turing machine is a kind of theoretical state machine. At any time, the machine is in any one of a finite number of states. It has an infinite tape that consists of adjacent cells as its unlimited

memory, with a tape head that reads and writes symbols and moves around on the tape, having the ability to store information.

On each cell is written a symbol. The symbols that are allowed on the tape are finite in number and include the blank symbol. Each Turing machine has its own alphabet, i.e. a finite set of symbols, which determines the symbols that are allowed on the tape. A Turing machine has a finite number of states and, at any point in time, the Turing machine is in one of these states. It begins its operation in the start state, and it halts when it moves into one of the halt states. In the next definition, we present a formal description of Turing machine.

Definition 2.17 (Turing Machine [Tur36, Sip13])**.** *A Turing machine T is a 7-tuple ($Q, \Sigma, \Gamma, \delta, s_0$, s_{accept}, s_{reject}) where Q, Σ, Γ and δ are all finite sets, and*

- *Q is the set of states.*
- *Σ is the input alphabet not containing the blank symbol $\sqcup$.*
- *Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$*
- *$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function, where $\{L, R\}$ is the direction of moving (left or right) on the machine's tape.*
- *$s_0 \in Q$ is the intial state.*
- *$s_{accept} \in Q$ is the accept state.*
- *$s_{reject} \in Q$ is the reject state, where $s_{reject} \neq s_{accept}$.*

Computing process: *At the start of computation, the machine is in the initial state q_0 and the head is positioned over a designated cell of the tape, the head is scanning exactly one cell of the tape.*

- *In a single step of computation, the machine looks up the symbol $a \in \Gamma$ where the head points.*
- *Check the current machine's state q and compute $\delta(q, a) = (q', a', d)$.*
- *Writes the symbol a' on the tape where the head is positioned, and sets the internal state to be q'.*
- *Moves the head one cell to the left or right according to the direction d.*

- *The computation stops, or halts, if the machine reaches a final state (an accepting or rejecting state), and is undefined if there is no valid transition and the current state is not final.*
- *The output is whatever is left on the tape, starting from the initial position and until the first blank.*

The above definition provides a description of Turing machines and how they perform computation. We can obtain equivalent definitions if more tapes are added, with different operation types such as read-only, write-only, bidirectional tapes, and using a different finite alphabet. These equivalent definitions are variants of the basic model, i.e. a *generalised model* of the basic one, which differs in their external configuration that makes this model robust and flexible, so we can adapt a generalised version of Turing machines.

2.6 Universal Machines

An important consequence of computability theory is that the set of computable functions is enumerable; each computable function can be assigned a natural number so that this number uniquely represents that computable function. This is due to the existence of an effective method of enumerating Turing machines, and that a Turing machine can compute these functions [CT06].

Turing machines can be enumerated by setting an encoding scheme that assign a unique identifier called the *Index*. The encoding method is represented by associating words over some coding alphabet.[7] Each Turing machine, say T can be computed with other Turing machines by including their indexes into T's input word. In essence, each Turing machine can have a tag m, which are ordered lexicographically, and each tag assigned a unique index i, indicating its location in this ordering. Using this idea of enumeration of his machines, Turing discovered an important fact about Turing machines, which is stated in the following.

Theorem 2.18 (Universal Turing Machine [AB09])**.** *Let T_i be the i^{th} item in a lexicographically ordered set of Turing machines such that $T_i \in \{T_1, T_2, ...\}$. There exists a Turing machine U such that for every $x \in \{0,1\}^*$, $U(x,i) = T_i(x)$.*

This proposition gives rise to the notion of *Universal Turing machine U*, which solves the problem of requiring different Turing machines, each one must be constructed for every new

[7] A suitable coding alphabet is $\{0,1\}$, because it is included in the input alphabet Σ of every Turing machine T [Rob15].

computation, for every input-output relation. The Universal Turing machine can solve this problem by having the ability to simulate any Turing machine.

The Universal Turing machine performs this simulation by set to receive a pair of inputs $\langle i, x \rangle$, build a machine T_i using i and then simulates this machine on the input string x, such that $U(\langle i, x \rangle) = T_i$. Therefore, the Universal Turing machine can imitate the behaviour of any other Turing machine. Based on the above; U can be viewed as a general-purpose computing device which receives two inputs, P and x, where P is considered as a program for U and x the input data for P [Pin07].

In this thesis we consider a type of Turing machines whose set of halting programs is prefix-free, i.e. the set of such programs form a prefix code. One reason to look at this type of machines is because no halting program is a prefix of another halting program.

Definition 2.19 (Prefix Turing machine [LV08])**.** *A prefix Turing machine T is defined as a Turing machine with one unidirectional input tape, one unidirectional output tape, and some bidirectional work tapes. Input tapes are read only, output tapes are write only, and in the unidirectional tapes the head can only move from left to right. The set of programs P on which halts on T forms a prefix-code. These programs are also called self-delimiting programs.*

We can define a Universal prefix Turing machine that simulates any Prefix Turing machine on an input, in a similar way to the ordinary Universal Turing machine [LV08].

2.6.1 Asymptotic Notation

Asymptotic notation can be used to express the approximate behaviour of a function when the argument tends towards a particular value or infinity with another function. In computational complexity theory, it used to categorise algorithms by how they respond, in terms of (computational) time and space resources, to different inputs. Instead of using the exact measuring of time and space, it is convenient to focus on the asymptotic behaviour of resources as approximate functions of the input size. There is a family of asymptotic notations with the same order of magnitude symbols: O, o, Ω and Θ which are defined formally as follows.

Definition 2.20 (Asymptotic Notations [Sip13])**.** *Let f and g be functions $f, g : \mathbb{N} \to \mathbb{R}$, then:*

- $f(x) = O(g(x))$ *if there exists a constant $c > 0$ such that $f(x) \leq c.g(x)$, for sufficiently large x,*

- $f(x) = o(g(x))$ *if* $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$ *i.e. for any constant* $c > 0$*, a number* n_0 *such that for all* $x \geq n_0$ *.*$f(x) < c.g(x)$*,*

- $f(x) = \Omega(g(x))$ *if there exists a constant* $c > 0$ *such that* $f(x) \geq c.g(x)$*, for sufficiently large* x*,*

- $f(x) = \Theta(g(x))$ *if both* $f(x) = O(g(x))$ *and* $f(x) = \Omega(g(x))$.

So, we use the notation $O(f(x))$ when we denote a function that does not exceed $f(x)$ by more than a fixed multiplicative factor. This is helpful whenever we want to simplify an expression by eliminating unnecessary detail, but also in case we cannot precisely estimate this quantity explicitly.

2.7 Kolmogorov Complexity

Kolmogorov complexity (also known as Kolmogorov-Chaitin or 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 [Kol65], R. Solomonoff [Sol64], and G. Chaitin [Cha66]. Intuitively, Kolmogorov complexity of a binary string is the length of the shortest binary program that is computed on a Universal Turing machine. Informally, it measures the information content, degree of redundancy, or the degree of regularities of a binary string.

Kolmogorov complexity tries to provide an answer to a fundamental question about randomness. Consider the following binary strings:

$$10101010101010101010101010101010101010101010101010$$

$$10011101001000101101101000101111010100100100110101$$

Comparing the above strings, it is easily to conclude that the second one is more random than the other. Applying the law of probability, each string has an equal probability of (2^{-50}) in being chosen at random from the all binary sequences. Therefore, probability does not explain the intuitive notion of randomness.

Kolmogorov complexity comes in handy at explaining the randomness or the level of patternlessness in strings. The notion of randomness is highly related to patterns in strings and how to describe these patterns. The first string in our example has a very short description e.g. (25 * 10) or

1 in position n iff n is odd, whereas the second one is difficult to describe in comparison to the first one. Kolmogorov complexity has advantages over the probabilistic based model when reasoning about randomness in programs and strings, which makes it the right approach for determining string complexity. Thus, given two strings, Kolmogorov complexity determines which string is more complex. In comparing two strings, x_1 and x_2, if the Kolmogorov Complexity of x_1 is less than the Kolmogorov complexity of x_2, then x_2 is more complex than x_1, because a larger program is required to describe x_2.

Unlike Shannon entropy (see Section 2.3.1), Kolmogorov complexity depends only on the string, and not on the probability distribution from which it is sampled, that is Kolmogorov complexity measures the intrinsic information and randomness of a given string [LV08]. Another important difference between entropy and the Kolmogorov complexity is that entropy measures the amount of information in the source regardless of the computational capabilities of the extractor. Despite these major differences, they are also very similar: they are measured in bits, and have some similar properties. It can even be shown that for a given distribution, the Kolmogorov complexity is asymptotically equal to the entropy of that distribution [CT06] (see Section 2.11).

There is a major drawback of Kolmogorov complexity: it is not computable, that is undecidable [CT06], even for using polynomial-time bounded Kolmogorov complexity, it takes an exponential time; however, methods have been developed to approximate it. Essentially, the Kolmogorov complexity of a binary string can be approximated using any (lossless) compression algorithm, and is then close to, but greater than the length of the ultimate compressed version of that string. This approximation corresponds to an upper-bound of the Kolmogorov complexity [LV08].

Definition 2.21 (Plain Kolmogorov Complexity [LV08])**.** *Let T be a Turing machine and $T(P)$ the output of T when it is given a program P. The Kolmogorov complexity of a binary string x with respect to T, C_T, is defined as follows*

$$C_T(x) = min\{|P| \mid T(P) = x\}$$

$C_T(x)$ is the minimal length of a program for T to compute output x. Obviously, the Kolmogorov complexity depends on the choice of the Turing machine T, and consequently that does not make the model robust. The presence of a Universal Turing machine (see Section 2.6) can solve this problem as the *Invariance Theorem* states; it basically shows the universality of Kolmogorov

complexity using a Universal Turing machine.

Theorem 2.22 (Invariance Theorem [CT06]). *Let U be a Universal Turing machine, for any other Turing machine T and for all binary strings x, there exist a constant c_T depending only on T, such that:*

$$C_U(x) \leq C_T(x) + c_T$$

This theorem helps to remove the dependence on a specific Turing machine and use instead a universal Turing machine. Therefore, Definition 2.21 can be redefined, according to the Invariance theorem, so the Turing machine T is replaced by a Universal Turing machine U. The Definition 2.21 can be extended to account for the case where an additional input is already available to the Universal Turing machine, when describing a string x, this case can be referred to as the *conditional plain Kolmogorov complexity*.

Definition 2.23 (Conditional Plain Kolmogorov Complexity [LV08]). *Let U be a Universal Turing machine. For any binary strings x, y, the Kolmogorov complexity of x given y:*

$$C_U(x \mid y) = min\{|P| \mid U(P, y) = x\}$$

2.8 Prefix Kolmogorov Complexity

Plain Kolmogorov complexity has some drawbacks. The plain complexity does not satisfy an important property, namely the *sub-additive* property: for two strings x, y, $C_U(x, y)$ is the length of the shortest program such that U computes both x and y, and in how many bits they differ, because normally x and y are fed to U in a concatenated form, i.e. $C_U(x, y) = C_U(\langle x, y \rangle) = C_U(x \parallel y)$. In this case, it is desirable to have $C_U(x, y) \leq C_U(x) + C_U(y) + O(1)$, but there is no information to guide U in order to locate the beginning and end of x and y in $x \parallel y$ with only $O(1)$. Prefix Kolmogorov complexity resolves this problem by using a prefix Universal Turing machine. As shown by [LV08], having prefix code running on a prefix Universal Turing machine we can identify x and y: $C_U(x, y) \leq C_U(x) + C_U(y) + 2\log(\min(C_U(x), C_U(y)))$, so the cost for U splitting $x \parallel y$ into a program for x and one for y is at most $2\log(\min(C_U(x), C(y))$.

Definition 2.24. *(Prefix Kolmogorov Complexity) The* Prefix Kolmogorov complexity $K(x)$ *of a*

binary string x with respect to a Universal prefix Turing machine U is defined as:

$$K(x) = min\{|P| \mid U(P) = x\}.$$

As we can see, the above definition is very similar to the plain Kolmogorov complexity in Definition 2.21, with one main difference: the presence of a prefix Universal Turing machine. The Invariance Theorem also holds for the prefix Kolmogorov complexity.

Similarly to the conditional plain Kolmogorov complexity, the conditional version of prefix Kolmogorov complexity $K(x \mid y)$ of x given y is the length of a shortest program P that computes x when y is given to P as input [GV04]. Formally, it is defined as follows.

Definition 2.25. *(Conditional Prefix Kolmogorov complexity) Let U be the prefix Universal Turing machine, and x, y binary strings. The prefix Kolmogorov complexity of x conditioned to y:*

$$K(x \mid y) = min\{|P| \mid U(P, y) = x\}$$

Unconditional Kolmogorov complexity is exactly equal to the conditional Kolmogorov complexity with an empty input such that $K(x) = K(x \mid \epsilon)$, where $U(P, \epsilon) = U(P)$. The conditional Kolmogorov complexity of x with x itself, as an input, is equal to zero i.e. $K(x \mid x) = 0$ [LV08].

The sub-addivity is ensured in the prefix version of Kolmogorov complexity as we explained earlier. Formally, the next theorem states this property.

Theorem 2.26 (Sub-additivity [LV08])**.** *Let x, y be binary strings,*

$$K(x, y) \leq K(x) + K(y) + O(1).$$

In the rest of the thesis, we will use the term Kolmogorov complexity and its notation K to refer to prefix Kolmogorov complexity that only works for prefix sets.

In the following we will provide some important properties of basic prefix Kolmogorov complexity. Prefix Kolmogorov complexity $K(x)$ is less than the length of a binary string x, and the conditional complexity $K(x \mid y)$ is less than the length of the original prefix Kolmogorov complexity of x. The following theorem states this fact.

Theorem 2.27 ([LV08]). *For all binary strings x and y*

$$K(x) \leq |x| + 2\log|x| + O(1) \quad \textit{and} \quad K(x\,|\,y) \leq K(x) + O(1).$$

The algorithmic information content (measured by Kolmogorov complexity) in an object such as a binary string and its conditional version, cannot be increased by any deterministic algorithmic method by more than a constant. The following two theorems illustrate this idea.

Theorem 2.28 (Information Non-Increase [She82, Tav11]). *For any recursive computable function $f : \{0,1\}^* \to \{0,1\}^*$ and for any binary string x, the Kolmogorov complexity of $f(x)$ is bounded by:*

$$K(f(x)) \leq K(x) + O(1)$$

Theorem 2.29 (Conditional Information Non-Increase [SUV14]). *Given a recursive computable function $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ and for any binary strings x, y, the Kolmogorov complexity of $f(x,y)$ is bounded by:*

$$K(f(x,y)\,|\,y) \leq K(x\,|\,y) + O(1).$$

The next theorem, known as *Muchnik's theorem* shows that there exists a shortest program P that converts a binary string y to another binary string x such that $|P| = K(x \mid y)$ and P is simple with respect to x, i.e. the level of dependence on y is small $O(\log n)$ for strings of length n [Muc02].

Theorem 2.30 (Muchniks theorem [Muc02]). *Let x, y be binary strings of length at most n. Then there exists a program p of length $K(x \mid y)$ such that $K(p \mid x) = O(\log n)$ and $K(x \mid \langle p, y\rangle) = O(\log n)$.*

There is a strong relation between random strings and Kolmogorov complexity. True random strings are incompressible; however, how random are strings, we need to answer. We can use Kolmogorov complexity to define randomness of strings, by introducing the notation of c-incompressibility.

Definition 2.31 (Incompressiblity [LV08]). *A binary string x is incompressible, if $K(x) > |x|$.*

Strings that are incompressible are patternless with lots of irregularities, because a patterned string requires a shorter description length than patternless strings. Intuitively, patternless sequences are

random, and *random sequences* can be used synonymously with incompressible sequence [LV08]. In the next section we quantify and measure the amount of randomness in binary string sets that are enumerable recursive.

2.8.1 Two Part Description

If we are aware that x belongs to a subset S of binary strings, then it could be easier to describe such strings. We do that by giving a description of S and then using the index of x in S, where the elements of S follow some ordering. This is called two-part description, which is formally stated as follows.

Lemma 2.32 (Two Part Description [LV08])**.** *Let S be an enumerable set of binary programs and $x \in S$, then:*

$$K(x) \leq K(S) + \log \#S + O(1)$$

The first part, $K(S)$, is the description of S; set S can be easily described. If S is a set of n bits strings then its descriptive complexity, i.e. a minimum program that generate that set, is at most of $O(\log n)$. The second part is the description of x's position within S.

We can also provide a conditional version of two-part description according to the following Lemma.

Lemma 2.33 (Conditional Two Part Description [Pin07])**.** *Let S be an enumerable set of binary programs and $x \in S$, for any binary string y we have:*

$$K(y \mid x) \leq K(S \mid x) + \log \#S + O(1)$$

Also, we can measure the level of randomness (irregularities) using randomness deficiency.

Definition 2.34 (Randomness Deficiency [LV08])**.** *The randomness deficiency of x with respect to a finite set of binary strings S containing x, $\delta(x|S)$, is defined as*

$$\delta(x|S) = \log \#S - K(x \mid S).$$

If $\delta(x|S)$ is large, then there is a description of x with respect to a set S that is considerably shorter than just giving x's position or index in S. An element x of S with low randomness

deficiency is said to be *typical*; the most efficient way to describe x using S is to find its exact location in the set.

2.8.2 Algorithmic Mutual Information

Algorithmic mutual information [GV04] measures the information that one object gives about another, i.e. the information that one sequence gives about another.

Definition 2.35 (Algorithmic Mutual information)**.** *Algorithmic mutual information of two binary strings x and y, $I_K(x;y)$, is given by*

$$I_K(x;y) = K(y) - K(y \mid x).$$

Mutual algorithmic information is non-negative ($\forall x, y. I_K(x;y) \geq 0$) and symmetric too [Gác74] up to a logarithmic additive term, as the following theorem states. For any binary strings x, y, z we write $K(x, y \mid z) = K(\langle x, y\rangle \mid z)$, and $K(x \mid y, z) = K(x \mid \langle y, z\rangle)$, where $\langle x, y\rangle$ means that x and y are joint input.

Theorem 2.36 (Algorithmic Chain Rule [Gác74])**.** *For all binary strings x, y, z*

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

Logarithmic factors like the ones needed in the previous theorem are pervasive in the theory of Kolmogorov complexity. As is done commonly in the literature, we mostly omit them in our results, making a note in the theorem statements that they are there.

Definition 2.37 (Conditional Mutual Information [LV08])**.** *The algorithmic mutual information of two binary strings x and y conditioned to a binary string z is defined as:*

$$I_K(x;y \mid z) = K(y \mid z) + K(x \mid z) - K(y, x \mid z)$$

2.9 Approximating Kolmogorov Complexity

Kolmogorov complexity is uncomputable due to undecidability of the halting program; however, it can be approximated based on losseless data compression as it shown in [KY96] Theorem 2 and in [LV08]. Intuitively, the theorem states that $K(x)$ is the lower bound [KY96] of all the compressions of x; therefore, we say that every compression $C(x)$ of x gives an estimation of $K(x)$.

Formally a decompression algorithm is considered as an algorithm D running on a Universal Turing machine U, such that for any binary string x

$$U(D(C(x))) = x$$

We say that a compressor is *Lossless* if there exists a decompresser that reconstructs the source message from the code message without losing any source-code data.

Many compression techniques can be used to approximate Kolmogorov complexity, such as *lempel Ziv 77* compression algorithm [ZL77], *Deflate(gzip)* [Sal06], *Burrows-Wheeler* transform (implemented in bzip2) [BW94] and *PPMD* compression (7-Zip) [CIW84]. The most popular compression technique among them is the Lempel-Ziv algorithm, which is dictionary-based scheme. Each repeated occurrence of a string is replaced with a pointer to the original occurrence. The pointer consumes less space than the string itself, provided that the matched string is sufficiently long enough [JF14]; the literals that are not matched will be produced verbatim.

Definition 2.38 (Compressor [CV05])**.** *A compressor is a lossless encoder mapping* $C : \{0,1\}^* \to \{0,1\}^*$ *such that the resulting code is a prefix code.*

For convenience of notation we identify a compressor with a code-word length function $C : \{0,1\}^* \to \mathbb{N}$, where $\mathbb{N}$ is the set of non-negative integers, i.e. the compressed version of a file x has length $C(x)$. The compressors we refer to here are bounded by $C(x) \leq |x| + O(\log|x|)$, where the logarithmic term is due to the compressed files being prefix code-words [CV05].

Cilibrasi et al. [CV05] proposed a set of axioms: *Idempotency, Monotonicity, Symmetry, Distributivity*, which defines the notion of a *normal compressor*. These axioms were used to ensure the desired properties of normalised compression distance (*NCD*) [CV05], because the outcome of *NCD*, i.e. how good is the measured distance result, greatly depends on how *normal* is the

compressor. In the following we write xy to express the concatenation $x \parallel y$ between two binary strings x and y.

Definition 2.39 (Normal Compressor [CV05])**.** *A compressor C is normal if for all binary strings $x, y, z \in \{0,1\}^*$ it satisfies the following axioms:*

- *Idempotency: $C(xx) = C(x)$, and $C(\lambda) = 0$, where λ is an empty string,*
- *Monotonicity: $C(xy) \geq C(x)$,*
- *Symmetry: $C(xy) = C(yx)$,*
- *Distributivity: $C(xy) + C(z) \leq C(xz) + C(yz)$,*

up to an additive term $O(\log n)$, where n is the maximum length of x, y and z.

A conditional version of compression was also defined in a similar way to Kolmogorov conditional and joint complexities (chain rule), there exists a binary string y such that

$$C(x \mid y) = C(xy) - C(y)$$

$C(xy)$ can be expressed as the excess number of bits in the compressed versions of xy compared to the compressed version of y. It was also shown, based on the distributivity property of normal compressor that conditional compressed information $C(x \mid y)$ satisfies the triangle inequality (see Definition 2.42).

$$C(x \mid y) \leq C(x \mid z) + C(z \mid y)$$

An important property that can be added to the above, is the *sub-additivity* which was proven too in [CV05]. It states that the sum of two compressed binary strings is lower bounded by the compression of their concatenation as the following:

$$C(xy) \geq C(x) + C(y)$$

2.10 Time-Bounded Kolmogorov Complexity

A way to make Kolmogorov complexity computable is the introduction of resource bounds. By giving the Universal Turing machine a limit on the allowed time or the spaced, we can search for all

programs within those bounds to describe a string x, and thereof, compute the resource-bounded Kolmogorov complexity of x.

Definition 2.40 (Time-bounded Kolmogorov complexity [LV08])**.** *Let $U^t(P, y)$ denote the output of U running a program P on binary string input y for at most $t(|P|)$ steps, where $t(n)$ is some function suitable for representing time constraints.*

1. *The* Time-bounded Kolmogorov complexity *$K^t(x)$ of a binary string x with respect to U is defined as:*

$$K^t(x) = min\{|P| \mid U(P) = x \text{ in less than } t(|P|) \text{ steps}\}.$$

2. *The Conditional Time-bounded Kolmogorov complexity relative to y is defined as:*

$$K^t(x \mid y) = min\{|P| \mid U(P, y) = x \text{ in less than } t(|P|) \text{ steps}\}.$$

Most of the the properties of $K(x)$ of binary string x also hold for $K^t(x)$. In particular, $K(x \mid y) \leq K(x) \leq |x| + 2\log|x| + O(1)$. Basically, $K^t(x)$ becomes close to $K(x)$ as $t(|x|)$ grows, similarly, the conditional version of $K^t(x \mid y)$ can be defined.

2.11 Kolmogorov Complexity and Shannon's Entropy

There is a direct relation between entropy and Kolomrogov complexity, which was investigated in [CT06] and [GV04]; these two notions turn out to be asymptotically the same. This is because the expected value for a random variable in strings of the Kolmogorov complexity is close to the Shannon entropy [LCL+04].

Theorem 2.41 ([GV04])**.** *Let X be a discrete random variable over a sample space $\mathcal{X} = \{0,1\}^*$. For any computable probability distribution f over $\mathcal{X}$, and entropy $H(X)$:*

$$0 \leq \sum_{x \in X} f(x)K(x) - H(X) \leq c_f$$

where c_f is a constant defined by $c_f = K(f) + O(1)$.[8]

[8] Note that f is a function instead of finite binary string, i.e. $K(f) = \min_i\{K(i) \mid$ Turing machine T_i computes $f\}$ [GV04].

The expectation of $K(x)$ differs from $H(X)$ by a constant depending on f as we see from the above theorem. However, Kolmogorov complexity, for a sufficiently large sequence, turns out to be approximately equal to the entropy [LV08], such that $H(X) \approx \sum_{x \in X} f(x)K(x)$.

2.12 Metrics and Measures

In mathematics, a number of concepts are defined using necessary and adequate sets of axioms. One of these concepts is the *metric*, which is called *measure of distance*. Krantz et al. [KLST71] show that a metric is a measure according to the representation theory of measurement.

Definition 2.42 (Distance Metric Axioms [KLST71])**.** *A distance is a function d with non-negative real values, defined on the Cartesian product $X \times X$ of a set X i.e. $d : X \times X \to \mathbb{R}^+$*

d is a metric on X if for every $x, y, z \in X$:

- $d(x, y) = 0$ *iff* $x = y$ *(identity axiom);*
- $d(x, y) + d(y, z) \geq d(x, z)$ *(The triangle inequality);*
- $d(x, y) = d(y, x)$ *(the symmetry axiom)*

If d is a metric, then (X, d) is called a metric space. A distance measure that satisfies the metric axioms is by definition a valid measure if it is described by proximity structure [KSLT89].

Definition 2.43 (Proximity Structure Axioms [KSLT89])**.** *A set X with quaternary ordering relations $\leq_X$: $\langle X \times X, \leq_X \rangle$, $=_X$: $\langle X \times X, =_X \rangle$, and $>_X$: $\langle X \times X, >_X \rangle$ is a proximity structure if and only if the following axioms holds, $\forall x, y \in X$:*

- $\leq_X$ *is a weak ordering of* $X \times X$*;*
- $(x, x) >_X (x, y)$ *iff* $x \neq y$ *(positivity);*
- $(x, x) =_X (y, y)$ *(minimality);*
- $(x, y) =_X (y, x)$ *(symmetry).*

If X is a proximity structure and (X, d) is a metric space, then according to the representation theory of measurement, d is a measure of distance (i.e. homomorphism from $\langle X \times X, \leq_X \rangle$ and $\langle \mathbb{R}, \leq \rangle$), iff:

$$\forall x, y, w, z \in X : (x, y) \leq_X (w, z) \iff d(x, y) \leq d(w, z)$$

We use the term metric and measure interchangeably to refer to the same thing, although they are different from a mathematical point of view. The measure is an outcome of the measurement process where a number is assigned to characterise a specific attribute (a feature or property of an entity).[9] Formally, it is defined as follows.

Definition 2.44 (Measure [KLST71])**.** *A measure μ is a mapping $\mu : A \to B$ which yields for every empirical object $a \in A$, a formal object (measurement value) $\mu(a) \in B$.*

[9] An entity may be an object, such as a person or a software specification, or an event, such as the testing phase of a software project.

3 Code Obfuscation

Code obfuscation is a software protection technique, which is used to obscure programs in order to deter an attacker that is equipped with reverse engineering and program analysis tools. This chapter will provide an overview of the current obfuscation definitions, threat model, the state-of-the-art techiques, and the methods to evaluate the security of code obfuscation.

3.1 Introduction

Different program analysis methodologies and techniques have been developed in order to provide automatic analysis of software behaviour. The main applications of program analysis are optimisation, correctness and determining the program properties. For example, optimisation allows compilers to produce optimised code that reduce redundancy and provides relatively safe usage of computation. As far as the correctness and type safety are concerned, the exact and absolute program analysis is impossible to achieve, typically due to undecidability of the Halting problem [Tur36] and Rice's theorem [Ric53]. Furthermore, it was shown that reverse engineering binary code using data disassembly and decompilation is undecidable [LD03]. Therefore, the ultimate aim of program analysis techniques is to provide an approximation to software behaviour through sound models, i.e. expecting program analysis to provide a larger set of possibilities than what will happen during runtime execution of the program [NNH15].

Code obfuscation was advanced as a software protection method to deter malicious attackers who are armed with program analysis tools and reverse engineering techniques. A determined attacker can eventually break the protected code and reverse engineer it [CTL97]. This was confirmed by Barak et al. [BGI+01] result, who showed that it is impossible, in general, to obfuscate. They construct an idealistic general purpose obfuscator based on a virtual black-box model (the obfuscated program should act as a black-box), then they showed that it is not possible for such

an obfuscator to exist. Furthermore, Appel [App02] proved under certain assumptions, which are related to how obfuscated programs behave, deobfuscation is NP-easy.[1]

Despite these results, the recent theoretical advances in code obfuscation theory by Garg et al. [GGH+13] showed the possibility of obfuscating programs using *indistinguishability* obfuscation, which is a relaxed version of virtual black-box obfuscation model. They construct *Multilinear Jigsaw Puzzles* to obfuscate programs, and proved them secure with respect to their indistinguishability obfuscation definition. However, these advances in obfuscation theory were only theoretical; so far, we have not seen any practical implementation of their proposed algorithm to obfuscate programs [BOKP15]. Nevertheless, the usefulness of code obfuscation rises from its potentials of delaying the exposure of software intellectual property for as long as possible time. The main aim of the current code obfuscation techniques is to make the process of program analysis and reverse engineering uneconomical for an attacker.

In this chapter we show the current threats to software such as program analysis and malicious reverse engineering, and present the untrusted host as an attack model for software. We introduce the notion of code obfuscation as the potential defence method against such attack. We also provide some of the theoretical background in this domain, in particular, the impossibility results of virtual black-box obfuscation and the latest advances that are based on a relaxed version of virtual black-box model, which provide positive results on code obfuscation. We provide an overview of obfuscating techniques based on Collberg et al. [CTL97] taxonomy on code obfuscation algorithms. We discuss some of deobfuscation methodologies that can be used to foil code obfuscation techniques. Finally, we present an overview of the current metrics, which are used to measure the quality of code obfuscation.

Chapter Layout: Section 3.2 shows the current threats to software, and their attack model. In Section 3.3 we provide the current advances in the theory of code obfuscation. Section 3.4 gives an overview of code obfuscation transformation methods. Section 3.5 discusses the obfuscation techniques according to their language paradigm. In Section 3.6 we provide an overview of the latest advanced techniques for code obfuscation. In Section 3.7 we explore the current deobfuscation techniques, which can be used to break code obfuscation algorithms. In Section 3.8 we discuss the different evaluation methods that are used to measure the quality of code obfuscation.

[1] NP-easy is the set of function problems that are solvable in polynomial time by a deterministic Turing machine with an oracle for some decision problem in NP [GJ90].

3.2 Threats to Software

Many different types of threats for software can be found; among these we identify two major threats to software: malicious reverse engineering and program analysis. Despite using reverse engineering and program analysis for legitimate reasons, they also present major issues for software security. Malicious reverse engineering and program analysis are considered the cornerstone for many different types of attacks on software, including: software piracy, tampering, unauthorised modifications, discovering vulnerabilities and exploit them.

3.2.1 Program Analysis

Program analysis tries to predict the dynamic behaviour of programs without running it. Precise analysis of program behaviour is impossible to achieve as the program input is unknown. If we do not know the value of the input, then it is very hard to identify which execution path it might take. Therefore, the answer can be only approximate. Approximation is an attempt to find an answer as close as possible to the precise behaviour of a program. Precision of an analysis is improved by reducing the amount of information describing spurious behaviour of the program. Approximation can be achieved through semantics-based static analysis which investigates the dynamic properties of program behaviour. Program analysis can be performed using two different approaches: static program analysis and dynamic program analysis.

Static program analysis. This type of analysis is conducted to examine and study the program properties without executing the code. Typical static program analysis techniques are: data flow, control flow, alias analysis, type analysis and abstract interpretations [NNH15]; in addition to program slicing, disassemblers, and decompilers. Static analysis is conservative, that means the properties that are found by static deobfuscating techniques are weaker than the ones that may actually be true (over-approximation). This guarantees soundness, although the induced properties may be so weak as to be useless.

Dynamic program analysis. Performed by testing the program on sample input data, since it is infeasible to test all possible program execution paths due to combinatorial explosion. Dynamic analysis precisely analyses only a subset of all possible execution paths or parts of a program, i.e.

this corresponds to an under-approximation [Pre07], examples of dynamic program analysis are dynamic testing such as fuzzing [SGA07], profiling, debugging and program tracing.

The Impossibility of Program Analysis

Some characteristics of program execution behaviour are difficult to model correctly [Ost77]. It is not possible to construct a procedure or analyser that will always, accurately, determine whether a specific path will be executed. The exact analysis is impossible due to the lack of knowledge of input data values. So the analysis can, at best, hope to yield information about a set of possible computations.

The difficulty of reasoning about program properties lies in the condition, that there are many paths in a program. However, not all paths correspond to an execution [Lan92]. In essence, the exact prediction of program behaviour requires an algorithm that terminates, which is the equivalent to the halting problem. For example, consider the following code[2], which is prone to 'divide by zero' errors:

```
read(x);
if (x > 0) then Y := 1;
else (Y := 0; S); // S is some other statement
z := 2 / Y; // error??
```

The issue in the above code is what value `Y` may have and whether this code could trigger the vulnerability, i.e. `Y=0`. Apparently `Y` may take 0 or 1, If `S` does not terminate, `Y` cannot be 0 and the zero bug situation will not be detected. However, since it is undecidable whether `S` terminates or not, we do not expect the analysis to detect this situation [NNH15].

3.2.2 Reverse Engineering

Reverse engineering techniques typically use program analysis tools, in order to perform the reverse process of reconstructing the program source code. It can be perceived as a methodology, which combines both static and dynamic program analysis tools.

The reverse engineering process consists of several stages that aim to produce the source code from the binary code. It starts with *disassembly* phase, where the machine code is translated to

[2]The pseudo code is equivalent, with some modification, to the code taken from Principles of Program Analysis by Flemming Nielson, Hanne Riis Nielson and Chris Hankin.

assembly code, then it ends up with the *decompilation* phase,[3] which rebuilds the higher level representations of the program from the assembly code. Typically, during this process, static program analysis techniques are employed first, followed as required by dynamic program analysis tools.

3.2.3 Threat Model for Software: Untrusted Host

Program analysis and reverse engineering are the typical attacks on software that are running on an untrusted host. This model of attack is widely known as *white-box model*, where the adversary or attacker has the host or the system under her/his full control. Aucsmith [Auc96] characterises three different levels for this model of attack :

1. The attacker uses standard debuggers and system diagnostic tools, no special analysis tools are required.

2. Specialised software analysis tools involved, such as specialised debuggers and sophisticated reverse engineering tools.

3. Specialised hardware analysis tools are employed, these tools include, for example, CPU emulators, and bus logic analysers.

White-box attacks are a very powerful type of attack especially for open computing platforms such as PCs. The attackers have unlimited access to program binaries; however, it is assumed that the attackers have very limited knowledge about the software source-code. Having the attacker operating in white-box model, does not rule out the possibility of subjecting the targeted software to *black-box* attack. In black-box model attacks, the software is considered as an *oracle*, because the attacker can only analysing the external behaviour of the software, where the internal knowledge of software is not required, or the attacker does not examine it.

Collberg and Nagra [CN09] propose an attack methodology which resemble attackers' behaviour or strategy during the attack process. The attacker goes through five phases : *black-box* phase, *dynamic analysis* phase, *static analysis* phase, *editing* phase, and *scripting* phase.

The attacker starts with black-box testing in order to reveal the external behaviour of the software; however this stage can be skipped if the attacker has a comprehensive understanding of

[3]The decompilation phase is discussed in more details in Section 7.3.7.

the software's functionality. Then the attacker moves to the dynamic analysis phase in order to gain more understanding of the internal working of the program. At this stage, the attacker has a high level understanding of the software and how it works. In the static analysis phase, the executable code is checked and investigated directly.

This covers the reverse engineering process that we discussed earlier. Now, the attacker is assumed to have very detailed understanding and a complete picture of the whole software design and its implementation. For example, the proprietary algorithms can be exposed, cryptography keys are revealed and vulnerabilities are discovered at this stage; therefore, the confidentiality of software is compromised. Reaching the editing phase, the adversary (attacker) uses the acquired knowledge of the software's inner work to modify its executable, or to integrate it with her/his own software for interoperability purposes.

So far the four stages are conducted manually, in practice these phases are not followed in order, they are interleaved with each other using a trial and error process, pattern matching, running and testing the code multiple times. Finally, when the attacker has fulfill her/his own goals and has enough confidence in the soundness of her/his work, a scripting code is written in order to automate this process.

3.3 Obfuscation Theory

In the previous sections we present the common threats and attack model for software, in the rest of the thesis we will focus on code obfuscation as a defence strategy to hamper the threats to software, in general, and malicious software engineering, in particular.

An obfuscator is a program or algorithm that transforms a program to another program in such a way that the transformed (obfuscated) code is functionally equivalent to the original one but more difficult to understand. The first attempt to define the notion of obfuscation was introduced by Collberg et al [CTL97], they define obfuscation in terms of semantics-preserving transformation functions.

Definition 3.1 (Collberg et al definition [CTL97])**.** *Let $P \xrightarrow{\tau} P'$ be a transformation of a source program P into a target program P'. The transformation $P \xrightarrow{\tau} P'$ is an obfuscating transformation, if P and P' have the same observable behaviour. More precisely, in order for $P \xrightarrow{\tau} P'$ to be an obfuscating transformation the following conditions must hold:*

- *If P fails to terminate or terminates with an error condition, then P' may or may not terminate.*
- *Otherwise, P' must terminate and produce the same output as P.*

3.3.1 The Impossibility of Obfuscation

Obfuscation aims to make the program or circuit unintelligible, while preserving its functionality. Ideally, an obfuscated program should be a virtual black-box (*VBB*), a form of strongest notion for code obfuscation security. In that sense, whatever an adversary can compute from an obfuscated program, it could also be computed from the input-output (oracle access) behaviour of the program [Had00], i.e. it should not leak information about the program except its input output behaviour.

Barak et al provided a formal definition of obfuscation in an attempt to achieve a well-defined security [BGI+01]. However they found a counterexample which showed this definition cannot always be met. They proved the existence of a set of programs or functions that are impossible to obfuscate according to that definition.

Preliminaries: *PPT* is a shorthand for probabilistic polynomial time Turing machine. Let $A^M(x)$ be the output of A when executed on input string x given oracle access to M where A and M are algorithms. Consider that $\mathcal{A}$ and $\mathcal{B}$ are probabilistic algorithms.[4] Given two inputs $\mathcal{X}$ and $\mathcal{Y}$, we denote by $\Pr[\mathcal{A}(\mathcal{X})] \simeq \Pr[\mathcal{B}(\mathcal{Y})]$, that is, the probability distributions of outputs of $\mathcal{A}$ and $\mathcal{B}$ on their inputs $\mathcal{X}$ and $\mathcal{Y}$ are the same with negligible difference. $|\mathcal{A}|$ is denoting the size of $\mathcal{A}$, where $\mathcal{S}^P(1^{|\mathcal{A}|})$, $\mathcal{S}$ is a (simulator) algorithm with input of length $|\mathcal{A}|$ and having an oracle access to a program P [5]. Let $\mathcal{O}$ be a probabilistic algorithm that takes an input x, and an additional input of random bits r, independent of x. Then the output of $\mathcal{O}$ depends on input x and random bits r.

Definition 3.2 (Vitual Black-box Obfuscator [BGI+01])**.** *Let $\mathcal{O}$ be an obfuscator and P a program, $\mathcal{O}(P)$ is an obfuscated program that must satisfy the following properties :*

- *Functionality: for any program P, $\mathcal{O}(P)$ and P compute the same function as $\mathcal{O}(P)$.*
- *Polynomial slowdown: for any Program P, the size and running time of $\mathcal{O}(P)$ are at most polynomially larger than the size and running time of P.*

[4] Probabilistic algorithms, contrast to deterministic algorithms, takes a source of random numbers which helps to produce random behaviour (random choices) for the algorithm, even for a fixed input.

[5] The oracle access to P, means we only have input-output access to P, rather than the P's internal or its source-code.

- *Virtual black-box: For any adversary $\mathcal{A} \in$ PPT (probabilistic polynomial time), there is a simulator $\mathcal{S} \in$ PPT, such that:*

$$\Pr[\mathcal{A}(\mathcal{O}(P))] \simeq \Pr[\mathcal{S}^P(1)^{|P|}]$$

The last property, virtual black-box, simply states that anything which can be efficiently computed from $\mathcal{O}(P)$ can also be computed given an oracle access to P.

Barak et al. proved obfuscation is impossible, in general, according to virtual black-box definition. They showed the existence of a set of functions $\mathcal{H}$ that are inherently unobfuscatable. Let $f \in \mathcal{H}$ be one of these functions, randomly chosen in $\mathcal{H}$, and has the property π on f such that: $\pi : f \to \{0, 1\}$. When given an oracle access to f, no algorithm exists that can efficiently compute $\pi(f)$, it just gives pseudo-random result. However given any algorithm A that computes f, $\pi(f)$ can be efficiently computed. This shows the virtual black-box property of perfect obfuscation is inherently flawed. This is a generalisation of Barak et al's result [BGI+01].

The significance of the Halting problem, in relation to program analysis and obfuscation, is the impossibility of providing an exact analysis of whatever property can hold for a program. However, this is not always the case in code obfuscation, as a program property can be found if the deobfuscator has access to the obfuscating code according to Andrew Appel's argument in [App02], which makes deobfuscation an NP easy. This is confirmed too in Barak's et al's results.

3.3.2 Indistinguishablity Obfuscation

As a consequence of the negative results of general-purpose code obfuscator, a more relaxed notion of obfuscation was proposed; Barak et al suggested another notion of program obfuscation called indistinguishability obfuscation: an indistinguishability obfuscator$i\mathcal{O}$ for a class of circuits c guarantees that, given two equivalent circuits c_1 and c_2 of the same size, that compute the same functionality, it is hard for a distinguisher $\mathcal{D}$ to differentiate between the distributions of obfuscations $i\mathcal{O}(c_1)$ and $i\mathcal{O}(c_2)$, i.e. they should be computationally indistinguishable to $\mathcal{D}$.

Definition 3.3. *(Indistinguishability Obfuscator). A uniform PPT machine $i\mathcal{O}$ is called an indistinguishability obfuscator for a circuit class $\{\mathcal{C}\}$ if the following conditions are satisfied:*

- *(Correctness). For all $c \in \mathcal{C}$, for all inputs x, we have that*

$$\Pr[c(x) = i\mathcal{O}(c(x))] = 1$$

- *(Indistinguishability). For any PPT $\mathcal{D}$, there is a negligible function α such that, for any two circuits c_1 and c_2 that compute the same function and are of the same size k, it holds that:*

$$\Pr[\mathcal{D}(i\mathcal{O}(c_1)) = 1] - \Pr[\mathcal{D}(i\mathcal{O}(c_2)) = 1] \leq \alpha(k)$$

The significance of using the indistinguishablity obfuscators, unlike the *VBB (virtual black-box)* obfuscator, is its ability to avoid the impossibility results. Recently, Garg et al. [GGH+13] proved the existence of general purpose obfuscation based on a candidate construction for indistinguishability obfuscation for all circuits. They proposed *Multilinear Jigsaw Puzzles* which are a simplified variant of multilinear maps. Garg et al's construction, and variants thereof, were shown to satisfy the *VBB* guarantee in ideal algebraic oracle models [BGK+14]. However, none of the aforementioned results proved possible in achieving *VBB* obfuscation in the plain model [BCC+14]. Moreover in [AS15], the authors present a framework for proving meaningful negative results on the power of indistinguishability obfuscation.

3.4 Code Obfuscation Transformation

This section gives a brief overview of some obfuscation transformation techniques, based, mostly on Collberg et al. [CTL97]. We will also consider other techniques that have been developed since then. We will start by discussing the main categories of obfuscation transformation: Lexical transformation, Control flow transformation and Data transformation.

3.4.1 Lexical Transformation

Lexical obfuscations are aimed at making the code unreadable. They are concerned with changing the layout of the program rather than its semantics. Lexical transformation is a one-way transformation, where the original formatting cannot be recovered. It alters the lexical structure of a program source code. Lexical transformations can be conducted by different ways; the most well known methods are: Remove Comments, Source Code Formatting and Scramble Identifiers.

Remove Comments. It is considered the simplest form of obfuscation. Comments are, normally, added by software developers as part of software's documentation. Removing a program's comments apparently makes the program less readable, although comments and debugging information are often removed during the compiling process.

Source-code Formatting. Changing the programs formatting by removing all the whitespace and indentations [Dra10] makes the program's source-code less informative to a reverse engineer. Examples of formatting obfuscation are found in the Obfuscation C contest [Dra10].

Scramble Identifiers. Identifiers like variables (including classes, methods, fields etc.) are changed in a confusing manner, for example 'Sum' variable is renamed as 'average'. Also, meaningful names such as 'Sum' or 'Output' are transformed into names such as 'd34' or '34g'.

3.4.2 Control Flow Transformation

This approach alters the flow of control within the program's code. Collberg et al [CTL97] provide an obfuscation catalogue for control flow transformation based on its potential effect. They categorise it into three main groups:

Aggregation. It breaks up computation that logically belongs together or merges computation that does not. Examples of aggregation transformation are inlining (replace method call with the body of the method), outlining (replace sequence of statements with a method call), interleaving (merge separate methods into one), cloning (create many copies of the same method) [CTL97] and loop transformation [Wol95].

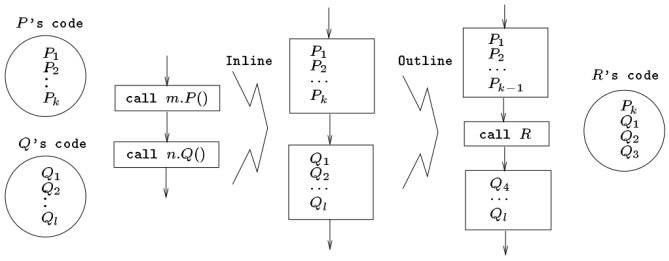

Figure 3.1: Inlining and outlining transformation [CTL97] .

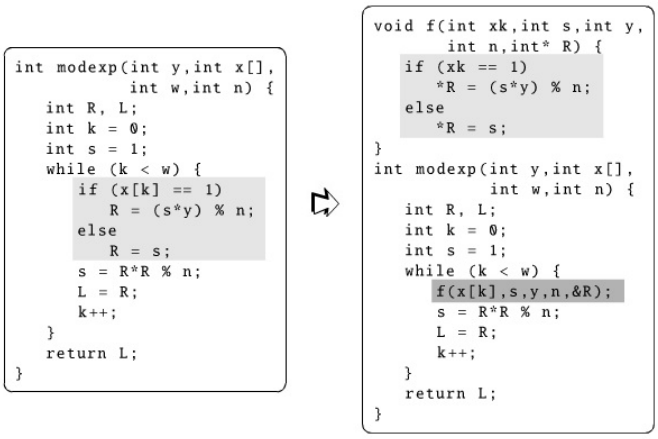

Figure 3.2: Code obfuscation by splitting a function into two functions [CN09].

Ordering. It can randomise the order in which the computations are carried out. For example, change the locality of terms with expressions, statements within basic blocks, methods within classes and classes within files. In some cases [CTL97] it is possible to reorder loops, for example by running the loops backward, loop reversal [Wol95] or control flow flattening. Control flow flattening removes the control flow structure that functions have, the nesting of loop and conditional statements, by flattening the corresponding control flow graph. Fig. 3.3 shows a modular exponentiation function commonly found in cryptographic algorithms, such as RSA. Each basic block is put as a case inside a switch statement and the switch is wrapped inside an infinite loop.

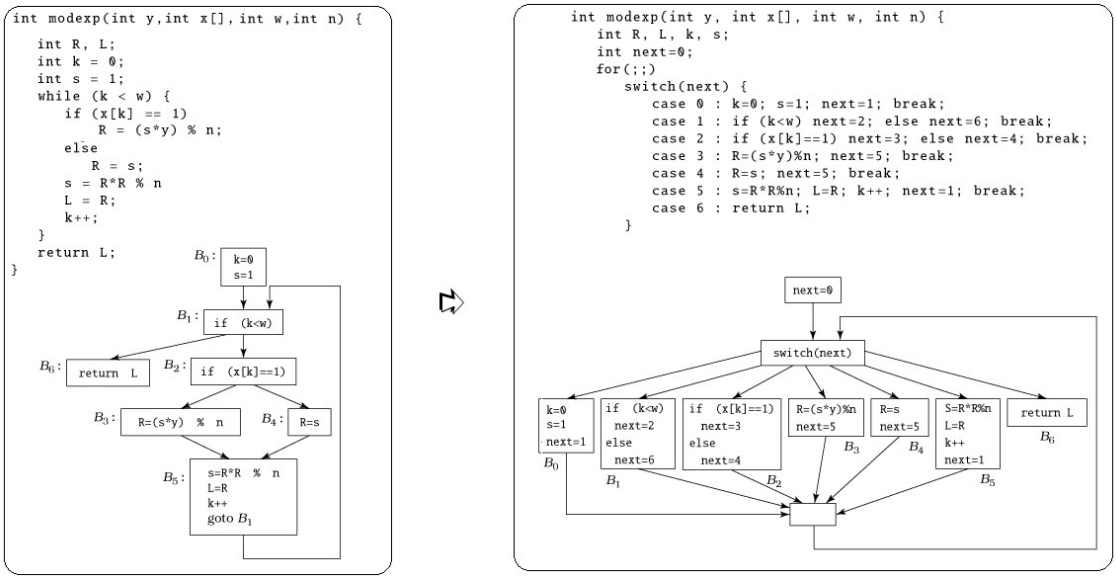

Figure 3.3: An example of code obfuscation using control flow flattening [CN09].

Computation. It makes algorithmic changes to the source code or inserts redundant, dead code or redundant operands (using algebraic laws in arithmetic expressions [CTL97]). The parallelisation of code is another important example of computational obfuscation: it obscures the actual flow of control by creating dummy processes that do nothing. It also splits a sequential section of code into multiple sections running in parallel. The significance of this obfuscating method is that the static analysis of concurrent processes is very difficult. The possible execution paths through the program grow exponentially with the number of executing processes or threads.

This form of obfuscation can be further sub-divided into two categories: dynamic dispatcher model [WDHK01] and opaque predicate [MTD06].

Dynamic Dispatcher Model. Wang et al [WDHK01] defined the Dynamic Dispatcher model based on the NP-Complete argument of determining the exact indirect addresses of dispatcher through aliased pointers. Chow et al [CGJZ01] on the other hand transform a program into a flat model; they proved their model to be PSPACE-Complete [6] to determine the reachability of a flattened program dispatcher. Control flow flattening makes the basic blocks look like having the same set of predecessors and successors [MTD06]. The actual control flow during execution is determined through the dispatcher. Therefore the dispatcher module is a crucial part in the flattened (obfuscated) program.

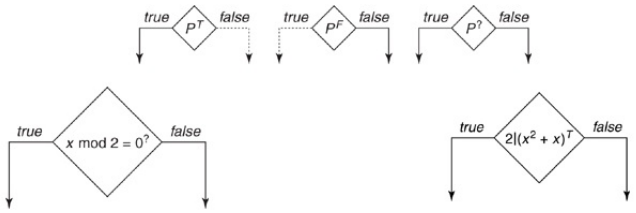

Figure 3.4: Examples of opaque predicate and fake paths [CN09].

Opaque Predicate. Opaque predicate is a conditional expression whose value is known to the obfuscator but is extremely difficult for an adversary to deduce statically. A predicate ϕ is defined to be opaque at a certain point p in a program, if its result is known at obfuscation time and only determined during runtime execution. The predicate is denoted by ϕ_p^T (ϕ_p^F) when it always evaluates to true (false) at program's point p.

[6] PSPACE is the set of all decision problems which can be solved by a Turing machine using a polynomial amount of space.

Opaque predicates can be used to create bogus code in programs, and it is used heavily in control flow obfuscations. There are several ways to create opaque predicates, such as *Algebraic predicates*, which consist of invariants that are based on number theory, for example, the opaque predicate of the mathematical expression: $\forall x \in \mathbb{Z} : 2 \mid (x^2 + x)$ is always true, as 2 divides $(x^2 + x)$. More complex examples are found in Fig. 3.5. Opaque predicates can also leverage the intractability property of pointer aliasing to construct *aliased* opaque predicates as in [CTL97]. The basic idea is to construct a dynamic data structure and maintain a set of pointers on this structure such as array. Opaque predicates can then be constructed using these pointers.

$\forall x, y \in \mathbb{Z}$	$: 7y^2 - 1 \neq x^2$
$\forall x \in \mathbb{Z}$	$: 3 \mid (x^3 - x)$
$\forall x \in \mathbb{N}$	$: 14 \mid (3 \cdot 7^{4x+2} + 5 \cdot 4^{2x-1} - 5)$
$\forall x \in \mathbb{Z}$	$: (2 \mid x) \vee (8 \mid (x^2 - 1))$
$\forall x \in \mathbb{N}$	$:2 \mid \lfloor \frac{x^2}{2} \rfloor$

Figure 3.5: Examples of number-theoretical true opaque predicates [Arb02]

3.4.3 Data Transformation

Data obfuscation targets the data structure in the program intending to obscure their usage and confuse their operations [ZHZ10]. Collberg et al [CTL97] classify the data transformation into storage and encoding, aggregation and data ordering. Drape [Dra10] provided a classification based on Collberg et al.'s taxonomy; in addition, he introduced the abstract data-type for obfuscation.

Variable Encoding. The basic idea behind variable encoding is to replace the variable by an expression, such that variable i is changed to $i = d \times i + e$ where d and e are constants. One of the main requirements of variable encoding is to be invertible, in the sense that the correct value of a variable can be obtained when it is required, especially once the variable output is needed for another variable. In short, this type of obfuscation depends on arithmetic computation that is equivalent to actual variable value. It is important to check the new value does not cause overflow e.g. integer overflow [Dra10]. The significance of this type of transformation is its effectiveness against program slicing [DMT07].

Merging and Splitting. In addition to obfuscating individual variables, two or more scalar variables can be merged into one variable, provided that the range of mixed variables fits within the

precision of the new obfuscated variable. For example, two 32-bit variables are merged into one 64-bit variable. Consider X and Y two 32-bit variables, merged into a 64-bit variable Z, then we can define Z as follows: $Z = N \cdot Y + X$ where N is a constant such that $0 \leq x < N$.

Splitting, on the other hand, can break up one variable in two or more variables, provided they are of fixed range such as boolean variables. Drape et al. [Dra10] demonstrates how an integer variable x can be split into two variables a and b such that: $a = x \div 10$ and $b = x \mod 10$.

Array Transformations. Arrays can be obfuscated by restructuring transformations such as changing the array indices. It is possible to split an array into several sub-arrays, merging two or more arrays into one, fold (increase the number of dimensions) or flattening an array (decrease the number of dimensions).

Abstract Data-types. The idea of obfuscating abstract data-types was introduced in Drape's PhD thesis [Ste04]. The abstract data-type consists of a unit containing declaration of the data type and the procedures that implement the data-type. This is in some sense similar to an object oriented view of classes, method and constructor. According to Drape's thesis the abstract data-types were specified using a functional language and their obfuscation was described according to data refinement [DREB98] principles.

A data obfuscation $\mathcal{O}$ can be specified by defining two functions: *cf*, the conversion function and *af*, the abstraction function. The conversion function performs the obfuscation and the abstraction function carries out the deobfuscation, which satisfy $cf : af \equiv skip$ where $\equiv$ denotes the equivalence between the outcome of *cf*:*af* to *Skip*. For example, a block of statements B is obfuscated to obtain $\mathcal{O}(B)$ using *cf* and *af* then $B = cf : \mathcal{O}(B) : af$.

3.5 Language Paradigm Obfuscation

Object Oriented Transformations. Many objected oriented languages rely on calls to standard libraries e.g. Java, however there is no way of obfuscating them [CTL97]. These library calls have to be implemented in the program itself to be easily obfuscated. Collberg et al [CTL97] provided two basic techniques that obfuscate object oriented languages through modifying inheritance relation among classes: splitting (refactoring) or inserting bogus classes. Refactoring is based on finding common elements or components of classes and moving them to new classes. Another

way of performing obfuscation is to use false refactoring on classes with no common behaviour or elements.

Intermediate Language Transformation. Obfuscation can be done on *Intermediate Languages (IL)* like Java byte-code and .Net CIL (Common Intermediate Language). The advantage of obfuscation on IL level is the ability to perform transformations that are not allowed on source-code level. For example jumps in loops are not possible in Java or C#, however it is allowed on IL level. Adding jumps in IL is referred to as irreducible jumps since it cannot decompile to source-code. Furthermore, many instructions in IL cannot directly decompile to a high-level language [Dra10]. It is very complicated to obfuscate IL code manually, especially obfuscated loops. Creating loop transformations at IL level is difficult, as we have to identify them in the code prior to obfuscation transformation. It is necessary to identify the loop header, the body and the exits of the loop; additionally it can be more complicated in the presence of nesting. Examples of obfuscation tools that are used to automated IL are DashO for Java and DotFuscator for .Net.

Exceptions. Exceptions change the control flow of programs. The exception handler can be used to obfuscate program; we can use opaque predicates with try-catch blocks. For example, if we have code consisting of two statements: `S1; S2;` then we can use the false opaque predicate ϕ^F to transform it into:

`Try {if (`ϕ^F`) {throw error} else S1;}`

`Catch (error) {/*dead code*/} finally {S2;}`

It is important to make sure that other uncaught exceptions are handled properly by different catch blocks, otherwise it causes incorrect behaviours or it may crash ungracefully [Dra10].

Pointers. Inserting pointers to code makes the deobfuscation process much harder, since conducting accurate alias analysis is considered NP-Hard [Hor97][BMRP09]. Wang et al [WDHK01] support their control-flow flattening techniques by using pointers. Also, pointers can be used to construct opaque predicates using complex dynamic pointer structures, such as double linked lists or binary trees.

3.6 Advanced Code Obfuscation Methods

In this section we present the most recent advances in code obfuscation techniques.

3.6.1 Preventive Obfuscation

Preventive obfuscation defends the obfuscated program against certain attack models [Ste04] especially against automatic reverse engineering techniques. The main goal of this approach is not to obscure readability and understanding of the program, instead to make the automatic deobfuscation difficult to perform. For example, Drape et al. [Ste04] create artificial dependences between program instructions in control-flow obfuscations to defend against attacks from program slicers.[7] Collberg et al [CTL97] described an interesting approach to preventive obfuscation by leveraging known problems and vulnerabilities in the current deobfuscators and decompilers, they called it *Target Preventive Transformation*.

3.6.2 Obfuscation Using Abstract Interpretation

Cousot and Cousot [CC77] formalise the relation between syntactic and semantic transformations within *abstract interpretation theory.* Abstract interpretation is a general theory for approximating the semantics of discrete dynamic systems [Cou96], where the program behaviour is seen as an abstract, approximation, of a concrete program semantics. According to abstract interpretation theory, given a concrete transformation it is always possible to find its abstract counterpart and vice versa. Concrete and abstract domains i.e. the poset of mathematical objects which the program runs, are related through a Galois connection (C, α, γ, D) where C is the concrete set of the programs, α is the abstraction function, γ is the concretisation function and D is the abstract set. Galois connection represents a particular correspondence (typically) between two partially ordered sets (posets).

Cousot and Cousot [CC02] introduced a semantics-based approach to formalise program transformations based on abstract interpretation. They establish a relation between syntactic and semantic transformation according to the abstract interpretation theory by mapping syntax to semantics abstractions.

[7] A program slicer is a program analysis technique, which produces a set of program's statements called slices. These parts (slices) of the program potentially affect the values computed at some point of interest.

Dalla Preda et al. [PG09] used Cousot and Cousot's framework [CC02] to provide a formalisation of program transformation to reason about obfuscation, by fixing a formal relation between syntactic and semantic obfuscation transformation. They investigated the potential of deriving semantics-based metrics for potency of code obfuscation. Shifting from the classic definition of code obfuscation, which is just about preserving the input-output of denotational semantics of program, they rely on preserving the code semantics in the hierarchy of abstract semantics [Cou97]. Potency becomes associated with the rate of abstraction of concrete preserved semantics. This means the transformed code is mapped to a lattice of abstract interpretation to measure the code potency.

3.6.3 Obfuscation Using Self-Modified Code

Self-modifying code is used to add additional layers of complexity to code obfuscation. However, it does not guarantee a provable level of obfuscation [MKP11] due to the shortage of relevant theoretical studies. The significance of self-modifying code for obfuscation is to reduce distinction between data and code. Static code does follow the convention of one-to-one mapping between instructions and memory addresses where self-modify code changes and mutates repeatedly during runtime execution.

Self-modifying code is highly effective against current static-analysis techniques; according to Bonfante et al. [BMRP09], this fact arises from the lack of thorough research on the theoretical aspects of self-modifying code, especially from a semantic point of view. That is why malware authors rely heavily on self-modifying code to avoid detection.

Self-modifying code has a long history of usage in software obfuscation. Early DOS programs used it to conceal their functionalities [GCK05]. Also, it was exploited to prevent programs from executing on competitor's operating system [BMRP09] by obscuring the code format and properly extracting and executing it at the runtime.

Self-modified code methods for obfuscation are usually guided by specific rules encoded by templates, either static or dynamic templates such as Madou's method [MAM+05]. Other techniques overwrite program code by dummy instructions using hiding and restoration such as Kanzaki's method [KMNM03].

3.6.4 Virtualisation-Obfuscated Software

Virtualisation-obfuscation implements a virtual environment and interpreter within which byte-code programs are executed. In virtualisation-obfuscation the original program's logic is inserted within the byte-code for a (custom) virtual machine (VM) interpreter. The language accepted by the interpreter is chosen at random at the time of protection. The interpreter itself is generated in accordance with the choice of language, and is also typically heavily obfuscated.

It is known that this kind of code obfuscation is very hard to be deobfuscated, at least from a practical point of view, due to the difficulty in recovering the logic of the original program. The analysis of the executed code can only reveal the logic of the byte-code virtual machine instead of the obfuscated program itself.

Virtualisation-obfuscators such as VMProtect[8] and Code Virtualizer[9] translate portions of the program's original x86 machine code into a custom language, which is then interpreted at run-time. The language interpreted by a virtualisation obfuscator tends to be similar to *RISC (Reduced Instruction Set Computing)*. *CISC (Complex Instruction Set Computer)* x86 instruction can be translated to multiple RISC-like virtual instructions. For example, for complex instructions such as

```
mov eax, [ebx+ecx*4+123456h]
```

the address calculation will be translated into several virtual instructions: one to get ecx, one to multiply ecx by 4, one to fetch ebx, one to add these quantities together, one to add 123456h to the result, and one to dereference the formed address [Rol09].

It is worth pointing out that this notion of obfuscation through virtualization is very similar to the idea of control flow flattening by Wang et al. [WDHK01]. In control flow flattening, the control flow of the program is altered in a two-step process. First, high-level control structures are transformed into `if-else-goto` constructs. Next, the `goto` statements are replaced with switch statements, where the switch variable is dynamically set within the `if-else` structure. The effect on the code is that any basic block in the control flow graph may be a predecessor or successor to any other basic block in the graph. The resulting analysis of the control flow then is transformed into a data flow problem. That is, identifying the value of the switch variable at each entry into the switch statement.

[8] http://vmpsoft.com/
[9] http://oreans.com/codevirtualizer.php

Code Virtualizer, for example, can generate multiple types of virtual machines with a different instruction set for each one. This means that a specific block of Intel x86 opcode can be transformed into different instruction sets for each machine, preventing an attacker from recognising any generated virtual opcode after converting from x86 instructions.

When an adversary attacks a block of obfuscated code by Virtualization obfuscation using decompilation, s/he cannot get the original x86 instructions. Instead, s/he will see a totally new instruction set which is not recognised by her/him or any other special decompiler. This will force the attacker to conduct exhaustive search trying to identifying how each opcode is executed and how each virtual machine works for each protected application. Which means that the attacker needs to deobfuscate the virtual machine, and then tries to figure out the obfuscated code. This approach is called *outside-inside*, which is extremely difficult, especially when the structure of interpreter is unknown.

3.7 Code Deobfuscation

Code obfuscation cannot provide a complete protection against malicious host attacks: a competent programmer, who is willing to invest enough time and effort, will always be able to reverse-engineer any obfuscated program. The process of attacking and removing the obfuscation transformations is called *deobfuscation*, it is considered as the reverse of code obfuscation transformation. Deobfuscation may include the use of any reverse engineering or program analysis tools, that we discussed in Section 3.2; in addition to any specialised tools to eliminate the obfuscated code and produce the original code, or an equivalent representation of this program. It has been shown in [UDM05], [CLD11a], and [YJWD15], that the combination of static and dynamic analyses, using a set of heuristics can disclose some of the significant obfuscating techniques.

There are limited studies which investigate rigorously and formally code deobfuscation, in a similar way to code obfuscation problem. One study by Della Preda et al. [Pre07] uses a static program analysis method based on abstract interpretation theory as a deobfuscator to break obfuscated code using opaque predicates. Kinder [Kin12] also applied abstract interpretation to deobfuscate visualization-obfuscated binaries.

Intuitively, an adversary with deobfuscation capabilities can be considered as a transformation of

an obscure program into a more intelligible program (easier to understand and work with), which is functionally equivalent to the original program [BF07]. The importance of defining and specifying a model for deobfuscation attacks, arises from the need to measure the quality of code obfuscation. To check whether or not a certain obfuscation technique is good at protected a program, it has to be certified against a certain attack model.

3.8 Obfuscation Evaluation

Evaluating the quality of code obfuscation necessitates the presence of robust metrics, in order to measure the complexity of the code. There were several attempts to provide concrete metrics for evaluating obfuscation [PAK98][AMDS+07]. Collberg et al. metrics [CTL97] are the most comprehensive in terms of comparing and contrast obfuscation transformation techniques. Collberg et al. used the software complexity metrics to evaluate obfuscation with respect to:

- **Cyclomatic Complexity [McC76]**. The complexity of a function increases with the number of predicates in the function.
- **Nesting Complexity [HM81]**. The complexity of a function increases with the nesting level of conditionals in the function.
- **Data-Structure Complexity [MK93]**. The complexity increases with the complexity of the static data structures that are declared in a program. For example, the complexity of an array increases with the number of dimensions and with the complexity of the element type.

These software complexity metrics allow us to formalise the concept of *potency* as a measure of transformation usefulness.

- **Potency.** The potency of transformation measures how more difficult the obfuscated code is to understand compared to the original code. Collberg et al [CTL97]l formalise it as follows:

$$Pt_{C_o}(P) = \frac{C_o(\mathcal{O}(P))}{C_o(P)} - 1$$

Where $\mathcal{O}(P)$ is the obfuscated version of program P using $\mathcal{O}$, $C_o(P)$ is the complexity of P, using a complexity metric C_o. $Pt_{C_o}(P)$, the potency of $\mathcal{O}(P)$ with respect to P, using

a complexity measure C_o, is a measure of the extent that $\mathcal{O}$ changes the complexity of P. $Pt_{C_o}(P)$ is a potent obfuscating transformation if $Pt_{C_o}(P) > 0$.

- **Resilience.** measures how well a transformation survives deobfuscation attack. Resilience is a qualitative measure reports on a scale of *trivial, weak, strong, full* and *one-way*, and takes into account the amount of time required to construct a deobfuscator, and the execution time and space actually required by the deobfuscator (polynomial time, exponential time).

- **Execution Cost.** measures the extra execution time and space for an obfuscated program $\mathcal{O}(P)$ compared with the original program P.

- **Quality:** combines potency, resilience and execution cost to give an overall measure.

- **Stealth.** an obfuscation is stealthy if it is not obvious in the obfuscated program, i.e. it resembles the original code as much as possible. Stealth is context-sensitive, what is stealthy in one program may not be in another one.

The first four properties are measured informally using qualitative scale: (for example, resilience measured on the scale of trivial, weak, strong, full, one-way). Stealth is difficult to be measured since it is context-sensitive, it depends on the whole program and also on the experience of the programmer [Dra10].

Anckaert et al [AMDS+07] proposed a quantitative framework to compare and evaluate code obfuscation techniques based on measuring four program's properties: *code, control flow, data* and *data flow*. This framework relied on the classical complexity measures, similarly to Collberg et al.'s work [CTL97], to measure their proposed properties. Every element of these properties is measured by using a specific classical complexity measure. Code property can be measured (depending whether its source or binary code) using the number of instructions, the number of operands, or the registers' values. Control flow, which resembles the possible execution sequences of program instructions, can be measured using Cyclomatic complexity or Nesting complexity. In data flow, which reflects the possible set of assigned numerical values at various points of program instructions, can be estimated, for example, using fan-in/fan-out complexity metric [HK81]. Fan-in/fan-out measures the information flow between program's components. The fan-in of a procedure is the number of modules or procedures that call this procedure; in addition to any data structures that are used by this procedure. The fan-out of a procedure are the number of modules and procedures that

are called by this procedure, and the data structures that are updated by the same procedure. Data that are related to programs can be measured using data-structure Complexity.

Linn et al proposed *confusion factor* [LD03] to evaluate the efficiency of code obfuscation with respect to disassemblers, i.e. measure the extent of disassembly's errors. The confusion factor measures the parts of the obfuscated program (instructions, basic blocks, or functions), which are mistakenly identified by disassembler as legitimate parts of the original unobfuscated code. It counts the number of instructions (also basic blocks, or functions) that are incorrectly disassembled, by finding their *diff* (difference). Formally, let A be the set of actual instructions addresses and A' is the set of disassembled instruction addresses. The confusion factor is computed as follows:

$$CF = \frac{|A - A'|}{A}$$

$|A - A'|$ denotes the count of incorrectly disassembled instructions. The confusion factor for functions and basic blocks are calculated similarly to the instructions addresses: a basic block or function is considered as being *inaccurately disassembled* if any of their instructions are incorrectly disassembled.

Tsai et al. [HW09] proposed a framework for quantitative analysis of *control flow graph (CFG)* obfuscating transformations. Their framework was based on a distance metric between the original and obfuscated program and code potency, and is restricted to control flow graph (CFG) obfuscation. The distance metric that they proposed is based on computing the *common sub-graphs (CS)* of the control flow graphs of two programs. Then, the number of edges in each common sub-graphs is counted to reflect the possible execution paths. Formally, given two CFGs G_P and $G_{\mathcal{O}(P)}$ of programs P and its obfuscated $\mathcal{O}(P)$ respectively, the graph distance is computed as follows:

$$d(G_P, G_{\mathcal{O}(P)}) = 1 - \sum_i \frac{2|edge(CS_i(G_P, G_{\mathcal{O}(P)}))|}{|edge(G_P)| + |edge(G_{\mathcal{O}(P)})|}$$

Where CS_i denotes the ith common sub-graph of G_P and $G_{\mathcal{O}(P)}$, and $edge(G)$ is the set of edges in CFG G.

3.9 Summary

Code obfuscation is one of the promising defence techniques for untrusted host based attacks. In this chapter we provided an overview of the theoretical backgrounds of code obfuscation, the impossibility results of virtual black-box obfuscation, and the positive results on code obfuscation using indistinguishablity obfuscator. We provide an overview of the current state-of-the-art for obfuscation techniques, deobfuscation techniques, and the current methods to measure the code obfuscation security.

4 Algorithmic Information Theory for Obfuscation Security

In this chapter we undertake a theoretical investigation of code obfuscation security and its threat model based on Kolmogorov complexity and algorithmic mutual information. We introduce a new definition that requires the algorithmic mutual information between a code and its obfuscated version to be minimum, allowing for a controlled amount of information to be leaked to an adversary. We argue that our definition avoids the impossibility results of Barak et al. and has an advantage over the obfuscation indistinguishability definition in the sense it is more intuitive and is algorithmic rather than probabilistic.

4.1 Introduction

Software developers strive to produce structured and easy to comprehend code, their motivation is to simplify maintenance. Code obfuscation, on the other hand, transforms code to a less structured and unintelligible version. It produces more complex code that looks patternless, with little regularity and is difficult to understand. We argue that irregularities and noise makes the obfuscated code difficult to comprehend.

Kolmogorov complexity [LV08] is a well known theory that can be used to measure regularities and randomness. Kolmogorov complexity is the basic concept of Algorithmic Information Theory, that in many respects adapts the viewpoint of well-established Information Theory to focus on individual instances rather than on random distributions. In general, Algorithmic Information Theory replaces the notion of probability by that of intrinsic randomness of a string.

Kolmogorov complexity is uncomputable; however it can be approximated in practice by lossless compression [KY96] [LV08], which helps to intuitively understand this notion and makes this theory relevant for real world applications. Our aim in this chapter is to provide a theoretical

framework for code obfuscation in the context of Algorithmic Information Theory: to quantitatively capture the security of code obfuscation, to discuss its achievability and to investigate its limitations and resilience against an adversary.

We introduce the notion of *unintelligibility* to define confusion in code obfuscation and argue that this is not good enough. We then propose our notion of security and compare both definitions. We argue that our model of security is fundamentally different from the virtual black-box model of Barak et al., and that because the impossibility result does not apply. Then we show that under reasonable conditions we can have secure obfuscation. We study the security of two main approaches to obfuscated code in software, *encoding* and *hiding*, at the sub-program level. Finally, we study the Kolmogorov complexity of applying multiple obfuscation transforms to a clear code, and the security case using our proposed definition.

Chapter layout: In Section 4.2 we discuss the intuitions behind our approach and propose the formal definitions of code obfuscation, and present positive results for secure code obfuscation against passive attackers. Section 4.3 studies the security of two main approaches to code obfuscation at the sub-programs level. Section 4.4 investigates the security case of applying multiple obfuscation to a clear code. Finally, we present a summary of the chapter.

4.2 Code Obfuscation using Kolmogorov Complexity

4.2.1 Motivation

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. 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. Human comprehension is a key in this case; an adversary has to understand the obfuscated code in order to recover the original; measuring code obfuscation has to take into consideration this human factor. Although measuring code comprehension is very subjective, there were some successful attempts to measure human cognitive reasoning based on Kolmogorov complexity [GZD11].

Code regularity (and irregularity) can be quantified, as was suggested in [Lat97] and [JF14], using

```
int n,i=0,x=0;
while(i<n)
 i=i+1
 x=x+i
```

(a) Sum code

```
int n, i=0; x=0; y=0;
while(i<n)
  i=i+1
if (7*y*y-1==x*x) //false
  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

```
int n; i=0; x=0; y=0;
while(i<n)
 i=i+1
 if (7*y*y-1==x*x)//false
  y=x*(i+1)
 else
  x=x+4*i
 if (x*x-34*y*y==-1)//false
  y=x*i
 else
  x=x-2*i
 if ((x*x+x)mod 2==0)//true
  x=x-i
 else
  y=x*(i-1)
```

(c) Three opaque predicates

Figure 4.1: 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 predicates and data encoding, which has less patterns and regularities comparing to (b).

Kolmogorov complexity and compression. Code regularity means a certain structure is repeated many times, and thus can be recognised. Conversely, irregularities in code can be explained as the code exhibiting different types of structure over the code's body. Regularities in programs were introduced by Jbara et al. in [JF14] as a potential measure for code comprehension; they experimentally showed using compression that long regular functions are less complex than the conventional classical metrics such as LOC (Line of Code) and McCabe (Cyclomatic complexity) could estimate.

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: s/he cannot provide a valid and brief, i.e. 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.

We demonstrate our motivation using the example in Fig. 4.1. We obfuscate the program in Fig. 4.1-(a) that calculates the sum of the first n positive integers, by adding opaque predicates with bogus code and data encoding. If we apply Cyclomatic complexity (McCabe [McC76]), a classical complexity measure, to Fig. 4.1-(b) the result will be 7. Cyclomatic complexity is based on control flow graph (CFG), and is computed by: $e - n + 2 \cdot p$, where e is the number of edges, n is the number

of nodes in CFG and p the number of connected components in graph (in our example $p = 1$). Fig. 4.1-(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 argue that we can find the short description of the program in Fig. 4.1-(b), due to presence of regularity by using lossless compression.

We take another obfuscated version in Fig. 4.1-(c) (of the same program); this code is obfuscated by adding three different opaque predicates. The patterns are less in this version comparing to Fig. 4.1-(b); where the Cyclomatic complexity is the same 7, and it does not account for the changes that occurred in the code. Assuming the opaque predicates of Fig. 4.1-(c) are equally difficult to break, attacking this code requires at least twice more effort than the code in Fig. 4.1-(b), as we need to figure out the value of two more opaque predicates. Furthermore, in Fig. 4.1-(b) the code can be compressed at higher rate than the code in Fig. 4.1-(c); again, this is due to the inherent regularity in Fig. 4.1-(b).

We argue that an obfuscated program which is secure and confuses an adversary will exhibit a high level of irregularity in its source-code and thus require a longer description to characterise all its features. This can be captured by the notion of Kolmogorov Complexity, which quantifies the amount of information in an object. An obfuscated program will have more non-essential information, and thus higher complexity, than a non-obfuscated one. Thus, we can use Kolmogorov complexity to quantify the level of confusion in obfuscated programs compared to the unobfuscated one.

4.2.2 Applying Kolmogorov Complexity to Code Obfuscation

In this section we present a novel approach for code obfuscation based on notions from Algorithmic Information Theory. We start with an intuitive definition that is inspired by practical uses of obfuscation. The rationale behind this definition is that an obfuscated program must be more difficult to understand than the original program. This uses the separate notion of c-unintelligibility:

Definition 4.1. *A program Q 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:

$$K(Q) \geq (c+1)K(P),$$

for some constant $c > 0$.

Definition 4.2. *A* c-Obfuscator $\mathcal{O} : \mathcal{P} \times \mathcal{L} \to \mathcal{Q}$ *is a mapping from programs with security parameters* $\mathcal{L}$ *to their obfuscated versions such that for all* $P \in \mathcal{P}, \lambda \in \mathcal{L}$. $\mathcal{O}(P, \lambda) \neq P$, *where* (P, λ) *is the input to* $\mathcal{O}$, *and satisfies the following properties:*

- ***Secrecy:*** *Security parameters are secret, and contains all the knowledge to obtain* P *given* $\mathcal{O}(P, \lambda)$.
- ***Functionality****:* $\mathcal{O}(P, \lambda)$ *and* P *compute the same function, such that* $[\![P]\!] = [\![\mathcal{O}(P, \lambda)]\!]$.
- ***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, *i.e. for some polynomial function* $p.|\mathcal{O}(P, \lambda)| \leq p(|P|)$, *and if* P *halts in* k *steps on an input* i, *then* $O(P, \lambda)$ *halts within* $p(k)$ *steps on* i.
- ***Unintelligibility****:* $\mathcal{O}(P, \lambda)$ *is c-unintelligible with respect to* P.

The security parameters $\mathcal{L}$ are the secret keys that are used to obfuscate programs, and contain all the knowledge that is required to obtain the original program from the obfuscated version. This is very similar to symmetric encryption, where the key is secret and is required for both encryption and decryption process.[1] On the other hand, this is different from public key encryption, the encryption key is public, where the decryption process requires a private key (secret key).

Now, it is interesting to ask to what extent unintelligibility is related to the quality of the security parameter λ. Is a large, i.e. long string, λ *necessary* for high unintelligibility? Is it *sufficient*?

We answer the first question in the positive by showing that c-unintelligibility sets a lower bound on the size of λ.

Lemma 4.3. *Consider a program* P *and an obfuscated version* $\mathcal{O}(P, \lambda)$ *with security parameter* λ *such that* $\mathcal{O}(P, \lambda)$ *is c-unintelligible with respect to* P. *Then,* $|\lambda| \geq cK(P) - O(1)$.

[1] There is a major difference between encryption and obfuscation: an adversary does not necessarily need to know λ to deobfuscate and obtain the original code.

Proof.

$$K(\mathcal{O}(P,\lambda)) \leq K(P,\lambda) + O(1) \qquad \text{by Theorem 2.28}$$

$$K(P,\lambda) + O(1) \geq K(\mathcal{O}(P,\lambda))$$

$$K(P) + K(\lambda \mid P) + O(1) \geq K(\mathcal{O}(P,\lambda)) \qquad \text{by Theorem 2.26}$$

By assumption on c-unintelligiblity, $K(\mathcal{O}(P,\lambda)) > (c+1)K(P)$, we have:

$$K(P) + K(\lambda \mid P) + O(1) \geq (c+1)K(P)$$

$$K(\lambda \mid P) \geq cK(P) - O(1).$$

By the basic property of Kolmogorov complexity: $|\lambda| \geq K(\lambda)$, and by Theorem 2.27: $K(\lambda) \geq K(\lambda \mid P)$. Therefore, $|\lambda| \geq cK(P) - O(1)$. □

To answer the second question, we show a counter-example. So far, we have not addressed the nature of $\mathcal{O}$ and how well it uses its obfuscation parameter. It could well be the case that $\mathcal{O}$ only uses some bits of λ to modify P. In an extreme case, it can ignore λ altogether and simply return $\mathcal{O}(P,\lambda) = P$. The result satisfies the first two properties of an obfuscator, but can be considered unintelligible only in the degenerate case for $c = 0$ and surely we would not call the resulting code obfuscated. Another extreme case is when $\lambda = P$, we would have at most $K(\mathcal{O}(P,\lambda)) \leq K(P) + K(\mathcal{O}) + O(1)$ which again would lead to a very small c. These two cases, although extreme, serve only to show that the quality of an obfuscator depends not only on λ but also on the obfuscation algorithm itself, $\mathcal{O}$. This is addressed later in Theorem 4.12.

Definition 4.2 is perhaps the first natural definition one can find, but it has one shortcoming. Merely requiring the obfuscated program to be complex overall does not mean that it is complex in all its parts, and in particular, that it hides the original program. To illustrate this point, consider the following example.

Example 4.4. *Consider an obfuscated program $\mathcal{O}(P,\lambda) = P \parallel \lambda$, which is a simple concatenation of P and λ. Define $n = |\mathcal{O}(P,\lambda)|$. We know $K(P \parallel \lambda) \simeq K(P,\lambda)$ within logarithmic precision (see [LV08] page 663). Then, by applying the chain rule of Theorem 2.36, $K(\mathcal{O}(P,\lambda)) = K(P \parallel \lambda) \simeq K(P) + K(\lambda \mid P) + O(\log n)$. For large λ independent of P, this might signify a large unintelligibility, but the original program can be extracted directly from the obfuscated version requiring only $O(\log n)$ to indicate where P ends and λ starts.*

This leads us to our second definition, where we require not that the obfuscated program be more complex than the original but rather, that it reveals almost no information about the original. This is captured by the notion of algorithmic mutual information and can be stated formally as:

Definition 4.5. *Consider a program P and its obfuscated program $\mathcal{O}(P, \lambda)$. We say $\mathcal{O}(P, \lambda)$ is a γ-secure obfuscation of P if the mutual information between P and $\mathcal{O}(P, \lambda)$ is at most γ, that is:*

$$I_K(P; \mathcal{O}(P, \lambda)) \leq \gamma.$$

We say $\mathcal{O}(P, \lambda)$ is a secure obfuscation of P if is γ-secure and γ is negligible.

It is common, in the literature about Kolmogorov complexity, to consider logarithmic terms negligible. Thus, if both P and $\mathcal{O}(P, \lambda)$ have lengths of order n, we might consider that $\mathcal{O}(P, \lambda)$ would be a secure obfuscation for $\gamma = \log n$. This intuition, however, is bound to fail in practice.

Programs are typically redundant, written in very well-defined and formal languages, with common structures, design patterns, and even many helpful comments. It is expected that the complexity of a non-obfuscated program be low, compared to its length. Consider then the case that for a given P and $n = |P|$ we have $K(P) = O(\log n)$, and a scenario similar to Example 4.4, where the obfuscation reveals the original program, then the mutual information between both programs is maximum, i.e.,

$$I_K(P; \mathcal{O}(P, \lambda)) = K(P) - O(\log n) = O(\log n).$$

Even though this obfuscation cannot be considered secure, the resulting mutual information is so small that Definition 4.5 would declare it secure. We have two ways out of this:

- we do not consider programs with $K(P) = O(\log n)$, since this is the error margin of the important properties of Kolmogorov complexity (see Section 2.7), and at this level we can not achieve significant results;
- or we consider a relative definition of security, requiring that the mutual information be only at most a negligible fraction of the information in P.

The second option leads us to the following definition:

Definition 4.6. *Consider a program P and its obfuscated program $\mathcal{O}(P, \lambda)$. We say $\mathcal{O}(P, \lambda)$ is a ϵ-secure obfuscation of P if the mutual information between P and $\mathcal{O}(P, \lambda)$ is at most $\epsilon K(P)$,*

that is:

$$I_K(P; \mathcal{O}(P, \lambda)) \leq \epsilon K(P),$$

for $0 \leq \epsilon \leq 1$.

We say $\mathcal{O}(P, \lambda)$ *is a secure obfuscation of* P *if it is* ϵ*-secure and* ϵ *is negligible in some appropriate sense.*

4.2.3 On the Impossibility of Obfuscation

There exist other definitions of obfuscation in the literature. Of particular importance to us is the work of [BGI+12], due to its famous impossibility result. As the authors argue in that paper, the black-box model they propose for obfuscation is too strong to be satisfiable in practice.

The black-box model considers a program[2] P obfuscated into $\mathcal{O}(P, \lambda)$ using λ if any property that can be learned from $\mathcal{O}(P, \lambda)$ can also be obtained by a simulator with only oracle access to $\mathcal{O}(P, \lambda)$.[3] This essentially states that $\mathcal{O}(P, \lambda)$ does not give any particular information about that property, since it is possible to learn it without having access to $\mathcal{O}(P, \lambda)$. Notice that this model does not compare an obfuscated program with an original one, but rather with its functionality.

This is different from the definitions that we have proposed so far. Our definitions can be used to capture this purpose, namely, measuring how much information a program $\mathcal{O}(P, \lambda)$ gives about the function [4] it computes, which is denoted by $[\![\mathcal{O}(P, \lambda)]\!]$.

It suffices to note that every function $[\![\mathcal{O}(P, \lambda)]\!]$ has a minimal program for it, say, Q. Then, its Kolmogorov complexity is the size of Q and for every other program $\mathcal{O}(P, \lambda)$ computing $[\![\mathcal{O}(P, \lambda)]\!]$ we have

$$K([\![\mathcal{O}(P, \lambda)]\!]) = |Q| \leq |\mathcal{O}(P, \lambda)|.$$

For every program P it must be that $K([\![\mathcal{O}(P, \lambda)]\!]) \leq K(\mathcal{O}(P, \lambda)) + O(1)$ by Theorem 2.28, otherwise we could build a program R of size $K(\mathcal{O}(P, \lambda))$ that produced $\mathcal{O}(P, \lambda)$ and then ran in succession R and $U(R)$.[5] This composition of programs is itself a program with complexity $K(\mathcal{O}(P, \lambda) + O(1)$ which would be smaller than the assumed minimal program $[\![\mathcal{O}(P, \lambda)]\!]$, i.e. a contradiction. Therefore, our definition can be changed to compare the obfuscated program with

[2] Or rather, a circuit or a Turing machine representation thereof.
[3] In oracle access we can only have access to the program's input-output.
[4] We assume all the functions that a program P is computing are decidable.
[5] That is, we run R to produce $\mathcal{O}(P, \lambda)$ then we execute the result of this first execution, that is $\mathcal{O}(P, \lambda)$ itself.

the *simplest* (minimal) program computing the same function.

Definition 4.7. *Consider an obfuscated program $\mathcal{O}(P,\lambda)$. We say $\mathcal{O}(P,\lambda)$ is ϵ-securely obfuscated if*

$$I_K([[\mathcal{O}(P,\lambda)]],\mathcal{O}(P,\lambda)) \leq \epsilon K([[P]])$$

for $0 \leq \epsilon \leq 1$.

We say $\mathcal{O}(P,\lambda)$ is a secure obfuscated program if it is ϵ-secure and ϵ is negligible.

We believe our definitions of obfuscation differ from the simulation black-box model in important ways, and because of this they avoid the impossibility result of [BGI+12].

Our definition is a less stringent form of obfuscation rather than a weak form of black-box obfuscation. We assume the functionality of an obfuscated program to be almost completely known and available to an adversary, and only require hiding the implementation rather than the functionality itself. This approach to obfuscation is very practical and pragmatic, especially for software protection obfuscation, as usually the customer likes to know exactly what a product does, although s/he might not care about how it was implemented.

In contrast, the black-box model definition requires that all properties of a given obfuscated program $\mathcal{O}(P,\lambda)$ must be hidden from any adversary that ignores its source-code, but has access to at most a polynomial number of points of $\mathcal{O}(P,\lambda)$. In the following we summarise the main differences between the Algorithmic Information Theory approach to obfuscation and the black-box approach.

1. We can see that in our case, the adversary knows more about the functionality. Since the functionality is mostly public, this would be equivalent to giving the simulator in the black-box model access to this extra knowledge, reducing the advantage of the adversary and possibly making some functions obfuscatable.

2. On the other hand, our definition allows the leakage of a small, but non-zero, amount of information. Compare this with the black-box model where a single-bit property that is non-trivially computed by an adversary renders a function un-obfuscatable. Our definition requires the adversary to be able to compute more than a few bits in order for obfuscation to be broken.

3. Our definition considers only deterministic adversaries, again making adversaries less powerful and reducing their advantage (see Chapter 5).

We can try to model the implications for our definition of a successful black-box adversary against obfuscation. Consider an adversary $\mathcal{A}^A$ attacking a predicate π in an obfuscated program $\mathcal{O}(P, \lambda)$ by accessing an algorithm A, such that $A(\mathcal{O}(P, \lambda)) = 1$ if and only $\mathcal{O}(P, \lambda)$ satisfies π. In this case, $\mathcal{A}^A$ is able to compute 1 bit of information about $\mathcal{O}(P, \lambda)$, and we want to measure how much this helps $\mathcal{A}^A$ in describing some simpler program P that implements the same function $[[\mathcal{O}(P, \lambda)]]$.

Since the adversary knows A, s/he can enumerate the set S, assuming S is a recursive enumerable set, of all programs that satisfy A. Then, for some program $P \in S$, we can use two-part description (see Section 2.8.1) and would have $K(P \mid \mathcal{A}^A) \leq K(S \mid \mathcal{A}^A) + \log \#S$.

Note that the set S may be infinite, and so enumeration is the best that can be done: we enumerate all relevant programs and run A with each of them, noting the result. We could try to avoid this infinity problem by noticing we are only interested in programs simpler than the original, and thus satisfying $K(P) \leq K(\mathcal{O}(P, \lambda)) \leq |\mathcal{O}(P, \lambda)|$. This does not give absolute guarantees, since in general there are programs with $|P| > |\mathcal{O}(P, \lambda)|$ and $K(P) \leq K(\mathcal{O}(P, \lambda))$, but our hope is that these are few and far between as they increase in length. Thus, even if we make this assumption and disregard a whole class of possibly relevant programs, we still have in the order of 2^n specimens. If $\mathcal{A}^A$ were a deterministic algorithm, we would have $K(S \mid \mathcal{A}^A) = O(1)$, and if S has less than a half of all possible programs, then indeed we would find that

$$\begin{aligned} K(P \mid \mathcal{A}^A) &\leq K(S \mid \mathcal{A}^A) + \log \frac{\#S}{2} \\ &= O(1) + \log \frac{2^n}{2} \\ &\leq n - 1 \end{aligned}$$

However, $\mathcal{A}^A$ is randomised, and in order to accurately produce S, for each program $Q \in S$, $\mathcal{A}^A$ must be run with a set of random coins r such that $\mathcal{A}^A(Q, r)$. One way to describe S from $\mathcal{A}^A$ would need a polynomial number of bits for each program in S. Now, we no longer have the comfort of a negligible $K(S \mid \mathcal{A}^A)$ term, and we can no longer be sure that knowing this property would in any way reduce the complexity of our target program.

We can still try to go around this problem, by allowing our enumerator to list not only all programs but also all possible random strings, and choosing the majority vote for each program. The length, and therefore the number, of possible random strings is bounded by the running time of the program, which in turn is bounded by a function of its length. Therefore, if we limit the length of our programs, then we limit the number of random strings to search.[6]

This would eliminate the necessity of considering the extra information due to the random coins, but on the other hand the running time would increase exponentially. Any successful adversary would be very different to the *PPT* adversaries of [BGI+01], and although our definition has been made, for ease of exposition, with unbounded Kolmogorov complexity (for unbounded adversaries), it is easy to change it to consider polynomially-bounded adversaries by using an alternative definition of mutual information (see Section 2.10):

$$I^*_K(P;\mathcal{O}(P,\lambda)) = K^{t()}(P) - K^{t()}(P\,|\,\mathcal{O}(P,\lambda)),$$

where $t()$ is a polynomial on the length of $\mathcal{O}(P,\lambda)$ that acts as a hard limit to the number of steps the universal machine can be run for. We address adversaries and their capabilities using Algorithmic Information Theory in Chapter 5.

With this definition of obfuscation, the above reasoning would lead to examples where non-obfuscatability by the black-box model does not prevent obfuscatability in the Algorithmic Information-Theoretic model. Our definition for security takes a program P (clear code), which supposedly is an easy and smart implementation of functionality $[\![P]\!]$, and compares it with $\mathcal{O}(P,\lambda)$, which is a different and supposedly unintelligible implementation of $[\![P]\!]$, so that the original P cannot be perceived or obtained from $\mathcal{O}(P,\lambda)$. The defenders' aim is not to prevent an adversary from understanding or finding $[\![P]\!]$, but to prevent her/him from finding their own implementations P.

This intuition best matches the idea of *best possible obfuscation* which was advanced by Goldwasser and Rothblum [GR07]. According to [GR07] an obfuscator $\mathcal{O}$ is considered as best possible if it transforms a program P so that any property that can be computed from the obfuscated program $\mathcal{O}(P,\lambda)$, can also be computed from any equivalent program of the same functionality. However, despite the close intuitive correspondence, our definition also differs from best possible obfuscation,

[6] Conversely, if we really want to enumerate all programs, then we also have to enumerate an infinite number of random strings.

in the sense that their definition relies on some form of black-box simulation. It was proved in [GR07] that best possible obfuscation has a strong relation with *indistinguishability obfuscation* (see Section 3.3.2), if $\mathcal{O}$ is an efficient obfuscation i.e. run in polynomial time.

An indistinguishability obfuscator is considered insecure if an adversary can distinguish between two obfuscated circuits (programs) (see Section 3.3.2). Indistinguishibility between two strings (programs) was studied by Laplante et al. [LR96] in the context of Algorithmic Information Theory, they proposed a modified version of Kolmogorov complexity that accounts for a distinguisher that differentiates between two binary strings. However, it is not clear, and remains an open problem whether their theoretical treatment can be used to model indistinguishability obfuscation, and under which conditions.

Indistinguishability obfuscation, even if it can be defined using Algorithmic Information Theory, is not very intuitive. Using indistinguishability obfuscation, it is very difficult to provide a guarantee about what obfuscation hides (just only if the distinguisher succeed); for example, if $\mathcal{O}$ produces a canonical representation of programs P_1 and P_2 without aiming to hide their implementation, then the indistinguishability definition is trivially satisfied too. Therefore, it does not provide a guarantee about what $\mathcal{O}$ can hide. On the other hand, our definition provides a security guarantee about what is hidden, due to obfuscation process, by comparing the original code to its obfuscated version using algorithmic mutual information.

4.2.4 Security and Unintelligibility

Our first attempt to characterise obfuscation was based on *unintelligibility*, and then we evolved to a notion of security based on *algorithmic mutual information*. The first notion seemed more immediately intuitive; traditional obfuscation techniques seem to rely only on making the code as complex as possible in an attempt to hide its meaning. But precisely this notion of *hiding meaning* is nothing more than reducing the information that the obfuscated program leaks about the original one, and so we believe the second approach to be the correct one. However, we can ask the natural question: is there any relation between these two concepts? Does high unintelligibility imply high security, or vice-versa?

We give a partial answer to this question. In certain situations, high unintelligibility will imply high security, as stated in the following theorem.

```
\\ variable that holds authentication password
string user-Input = input();
string secure-password = ...;
if secure-password ==  user-Input
 grant-access();
else
 deny-access();
```

P : simple password checker

```
string O@ = x0();
string $F=...;
if O@== $F
 x1();
else
 x2();
```

$\mathcal{O}(P,\lambda)$: obfuscating P

Figure 4.2: Obfuscating a password checker program

Theorem 4.8. *Consider a program P of length m and its obfuscated version $\mathcal{O}(P,\lambda)$ of length $n > m$ (where n is at most polynomially larger than m), satisfying c-unintelligibility for $c \geq \frac{n}{K(P)}$. Assuming that the obfuscation security parameter λ satisfies $K(\mathcal{O}(P,\lambda)\,|\,P) \geq K(\lambda\,|\,P) - \alpha$, for some $\alpha \in \mathbb{N}$, then up to $O(\log n)$:*

$$I(P;\mathcal{O}(P,\lambda)) \leq \alpha$$

Proof. $K(\mathcal{O}(P;\lambda)) \geq (c+1)K(P)$ by c-unintelligibility assumption, and by Theorem 2.28, $K(\mathcal{O}(P,\lambda)) \leq K(P,\lambda) + O(1)$[7], we have:

$$\begin{aligned} K(P,\lambda) &\geq (c+1)K(P) \\ K(P,\lambda) - K(P) &\geq cK(P) \\ K(\lambda\,|\,P) &\geq cK(P) \qquad \text{by Theorem 2.36} \end{aligned}$$

[7]The $O(1)$ term is absorbed by the logarithmic additive term that we are not notating.

Now, for mutual information, we have

$$
\begin{aligned}
I(P;\mathcal{O}(P,\lambda)) &= K(\mathcal{O}(P,\lambda)) - K(\mathcal{O}(P,\lambda)\,|\,P) && \text{by Definition 2.35}\\
&\leq K(\mathcal{O}(P,\lambda)) - K(\lambda\,|\,P) + \alpha && \text{by } c\text{-unintelligibility assumption}\\
&\leq n - cK(P) + \alpha\\
&\leq n - n + \alpha && \text{by assumption}\\
&= \alpha
\end{aligned}
$$

□

Intuitively, if we consider an optimal obfuscation security parameter (it has all the information needed to produce $\mathcal{O}(P,\lambda)$ from P, but not much more than that), we can say that if $\mathcal{O}(P,\lambda)$ is c-unintelligible for large enough c, then $\mathcal{O}(P,\lambda)$ is a secure obfuscation of P.

The above theorem shows when high c-unintelligibility implies security of code obfuscation. It turns out that the reverse implication does not exist, as the following theorem illustrates.

Proposition 4.9. *There exists an obfuscation function $\mathcal{O}$ that produces arbitrarily secure programs, according to Definition 4.6, and yet does not satisfy c-unintelligibility for $c \geq 0$.*

Proof. Consider first the case of program P of Fig. 4.2, that simply checks a password for access, and its obfuscated version $\mathcal{O}(P,\lambda)$, which was computed using layout obfuscation [CTL97]: variable and function renaming and comment deleting. The obfuscated variable and function names are independent of the original ones and so the information that $\mathcal{O}(P,\lambda)$ contains about P is limited to the unchanging structure of the code: assignment, test and `if` branch. The complexity of the original code can be computed from several independent parts of the program: the complexity of comments: n_c, the complexity of structure: n_s, the complexity of variable and function names: n_v, and therefore we can write $K(P) = n_c + n_s + n_v$.

The only thing that $\mathcal{O}(P,\lambda)$ can give information about is the structure part, since all the other information was irrevocably destroyed in the process: there is no data remaining that bears any relation to the lost comment or the lost function names. Therefore,

$$I_K(P;\mathcal{O}(P,\lambda)) \leq n_s = \frac{n_s}{n_c + n_s + n_v} K(P).$$

We can make the fraction $\frac{n_s}{n_c+n_s+n_v}$ as small as necessary by inserting large comments, long names and representing structure as compactly as possible, for example, keeping names in a dictionary block and indicating their use by pointers to this. However, the complexity of$\mathcal{O}(P,\lambda)$ is less than that of P, since there is less essential information to describe: the same structure, no comments and the function names could be described by a simple algorithm. Therefore, we have that c-unintelligibility can not be satisfied for any non-negative value of c. This shows that high ϵ-security does not imply high unintelligibility. □

The next theorem shows how we can obtain a certain level of security that depends on: how complex is the obfuscated code and the obfuscation key (security parameter), and the level of dependency between the original program and the obfuscation key.

Theorem 4.10. *Let $\mathcal{O}(P,\lambda)$ be an obfuscated program for a program P of length n and λ be independent and random obfuscation security parameter, satisfying $K(\lambda) \geq n - \alpha$ and $K(P,\lambda) \geq K(P)+K(\lambda)-\beta$, for some $\alpha,\beta \in \mathbb{N}$. Suppose the obfuscation $\mathcal{O}(P,\lambda)$ satisfies $K(\mathcal{O}(P,\lambda)\,|\,P) \geq K(\lambda\,|\,P) - O(1)$. Then up to a logarithmic factor:*

$$I_K(P;\mathcal{O}(P,\lambda)) \leq \alpha+\beta$$

Proof.

$$\begin{aligned}
I_K(P;\mathcal{O}(P,\lambda)) &= K(\mathcal{O}(P,\lambda)) - K(\mathcal{O}(P,\lambda)\,|\,P) && \text{by Definition 2.35}\\
&\leq K(\mathcal{O}(P,\lambda)) - K(\lambda\,|\,P) && \text{by assumption}\\
&\leq K(\mathcal{O}(P,\lambda)) - K(\lambda,P) + K(P) && \text{by Theorem 2.36}\\
&\leq n - K(P) - n + \alpha + \beta + K(P) && \text{by assumptions}\\
&\leq \alpha+\beta
\end{aligned}$$

□

The first two assumptions are natural: picking λ at random and independently of P will satisfy high complexity and low mutual information with high probability, so we can simply assume those properties at the start. The third assumption ($K(\mathcal{O}(P,\lambda)\,|\,P) \geq K(\lambda\,|\,P) - O(1)$), however, is not immediately obvious: it describes a situation where the obfuscation key λ is *optimal*, in the sense

that it contains just the amount of information to go from P to $\mathcal{O}(P,\lambda)$, within a small constant term. The following lemma shows how λ must have a minimum complexity to allow the derivation of P to $\mathcal{O}(P,\lambda)$.

Lemma 4.11. $K(\lambda \mid P) \geq K(\mathcal{O}(P,\lambda) \mid P) - O(1)$

Proof. Given the obfuscator $\mathcal{O}$ and any λ, construct the function $q_\lambda(\cdot) = \mathcal{O}(\cdot,\lambda)$, and let Q_λ be a program that implements it. Then, clearly, $U(Q_\lambda, P) = \mathcal{O}(P,\lambda)$ and so $|Q_\lambda| \geq K(\mathcal{O}(P,\lambda) \mid P)$. To describe Q_λ, we only need to specify λ and instructions to invoke $\mathcal{O}$ with the proper arguments, but since P is already known, we only need to find a shorter description for λ. This gives $K(\mathcal{O}(P,\lambda) \mid P) \leq K(\lambda \mid P) + O(1)$. □

An optimal obfuscation key λ is then the one that uses as little information (minimum complexity) as possible in the obfuscation function, to go from P to $\mathcal{O}(P,\lambda)$.

Obfuscation techniques can use randomness or not. In Proposition 4.9, we showed one case where names could be obfuscated in a deterministic way, without any randomness. However, we could equally have used instead random names, with the same effect on security but increasing unintelligibility as well. We can as well consider that the set of obfuscation techniques is finite and describe each of them by a unique number. This way, we can characterise a single application of obfuscation by a key composed of the technique's index and the randomness needed.

We now proceed to show that it is possible to achieve obfuscation security according to our definitions, but restricted to a passive adversary, that is, one that does not realise transformations over the intercepted code. The intuition is to use obfuscation techniques that behave as much as possible as secure encryption functions, namely, using random keys (security parameters) that are independent from the code and large enough that they obscure almost all original information. The crucial difference that enables security is that because an obfuscation technique preserves functionality, we do not need to deobfuscate the obfuscated code and so we do not need to hide the key.

First, we prove the effect of obfuscating an elementary piece of code, by application of a single obfuscation technique. Then, we reason about the case of a full program, composed of several of these independent blocks.

Theorem 4.12. *Let p represent a program block, $\mathcal{O}$ an obfuscation technique and $\lambda \in \mathcal{L}$ an obfuscation key with a fixed length n. Let $\mathcal{O}(p,\lambda)$ be the obfuscated block. Assume $\mathcal{O}$ is such*

that it produces an output with length $\ell \leq n + \gamma$, where γ is the extra length due to obfuscation, and is "nearly-injective" in the following sense: for every p, any subset of $\mathcal{L}$ of keys with the same behaviour for p has cardinality at most polynomial in n. That is, for all p and $\lambda_0 \in \mathcal{L}$, $|\{\lambda \mid \mathcal{O}(p,\lambda) = \mathcal{O}(p,\lambda_0)\}| \leq n^k$, for some positive integer k.

Then, if the key is random, $K(\lambda) \geq n - \alpha$ and independent from p, $K(\lambda \mid p) \geq K(\lambda) - \beta$, for some $\alpha, \beta \in \mathbb{N}$, the obfuscated code $\mathcal{O}(p,\lambda)$ is $(\alpha + \beta + \gamma)$-secure up to a logarithmic term.

Proof. By symmetry of information Theorem 2.36, we can write

$$K(p \mid \mathcal{O}(p,\lambda)) = K(\mathcal{O}(p,\lambda) \mid p) + K(p) - K(\mathcal{O}(p,\lambda)).$$

It is easy to see by Theorem 2.29 that $K(\mathcal{O}(p,\lambda) \mid p) \leq K(\lambda \mid p) + O(1)$, since $O(1)$ is a constant independent of p, λ or $\mathcal{O}(p,\lambda)$ it can be ignored for a sufficient large size programs [CT06]. As well, we can show the reverse inequality. To produce λ from p, we can first produce $\mathcal{O}(p,\lambda)$ from p (with a program that takes at least $K(\mathcal{O}(p,\lambda) \mid p)$ bits) and then build the set, $S_{p,\mathcal{O}(p,\lambda)} = \{\lambda \mid |\lambda| \leq n, [\![\mathcal{O}(p,\lambda)]\!] = [\![p]\!]\}$ of all compatible λ, by a program whose length is $O(1)$. Finally, we just have to give the index of λ in this set, and so $K(\lambda \mid p) \leq K(\mathcal{O}(p,\lambda) \mid p) + \log \# S_{p,\mathcal{O}(p,\lambda)}$ by Lemma 2.33. Then,

$$\begin{aligned} K(\mathcal{O}(p,\lambda) \mid p) &\geq K(\lambda \mid p) - \log \# S_{p,\mathcal{O}(p,\lambda)} \\ &\geq K(\lambda) - \beta - \log \# S_{p,\mathcal{O}(p,\lambda)} && \text{by assumption} \\ &\geq n - \alpha - \beta - \log \# S_{p,\mathcal{O}(p,\lambda)} && \text{by assumptions on } \lambda. \end{aligned}$$

By assumption on the output of $\mathcal{O}$, $K(\mathcal{O}(p,\lambda)) \leq |\mathcal{O}(p,\lambda)| \leq n + \gamma$, and by Theorem 2.36, we have:

$$\begin{aligned} K(p \mid \mathcal{O}(p,\lambda)) &\geq K(p) + n - \alpha - \beta - O(\log n) - K(\mathcal{O}(p,\lambda)) \\ &\geq K(p) - \alpha - \beta - \gamma - O(\log n) \end{aligned} \tag{4.1}$$

Therefore,

$$\begin{aligned} I_K(p; \mathcal{O}(p,\lambda)) &= K(p) - K(p \mid \mathcal{O}(p,\lambda)) && \text{by Definition 2.35} \\ &\leq K(p) - K(p) + \alpha + \beta + \gamma + O(\log n) && \text{by Eq. (4.1)} \\ &\leq \alpha + \beta + \gamma + O(\log n) \end{aligned}$$

□

The randomness and independence conditions for the keys are natural. The other two conditions may seem harder to justify, but they ensure that $\mathcal{O}$ effectively mixes p with the randomness provided by λ: the limit on the size of subsets of $\mathcal{L}$ implies a lower bound on the number of possible obfuscations for p (the more the better); on the other hand, the limit on the length of the output of $\mathcal{O}$ forces the information contained in p to be scrambled with λ, since it can take only a few more bits than those required to describe λ itself. The extreme case is similar to One-Time Pad encryption[8], when both the output and λ have the same size n, and $\mathcal{O}$ is injective for each p: there are 2^n keys, as well as possible obfuscations for each p. Furthermore, because the obfuscated code has the same length as λ, the exact same obfuscated strings are possible for each p, maximizing security.

In general, a program is composed of several of these minimal blocks in sequence, regardless of the execution flow (control flow). The above proof shows when the obfuscated block $\mathcal{O}(p_0, \lambda)$ gives no information about its original block, say p_0. As well, $\mathcal{O}(p_0, \lambda)$ cannot give any more information about any other block p_1, as there is no relation between p_1 and $\mathcal{O}(p_0, \lambda)$. At best, there is some information in p_1 about p_0, but then the information given by $\mathcal{O}(p_0, \lambda)$ about p_1 should be at most that given by about p_0. Therefore, we conclude that if all the sub-blocks in a program are securely obfuscated, then the whole program is. The above theorem, hence, shows that secure obfuscation is possible under very reasonable assumptions.

4.3 Individual Security of Code Obfuscation

Studying the security of individual instances of obfuscated code provides more granularity. Even if the obfuscated program is considered secure according to our definition, it may have parts (sub-programs) which can provide some information about other obfuscated parts, which reduces the security of the obfuscated code. It could be that a program is obfuscated but some modules are not: some part of the obfuscated code stays still in its original form. We can demonstrate this relation by providing some bounds on the complexity of sub-programs in the same program.

We can view (obfuscated) programs as finite, and therefore recursively enumerable, sets of sub-programs (blocks or modules) such that $Q = \{q_i \mid i \in \mathbb{N}\}$. If Q is an obfuscated program then it may consist of obfuscated and non-obfuscated (clear) modules, it could be the case that the

[8] Which is proved to be an unconditionally secure symmetric cypher.

obfuscation process left some modules of code unobfuscated, that is: there are q_j, q_i in Q, where q_i is an obfuscated module and q_j is a non-obfuscated module.

Theorem 4.10 demonstrates the effect of security parameters λ on the whole obfuscation process. The following results show the effect of each individual security parameter on each obfuscated sub-program. Choosing a good λ (with a good source of randomness) requires the minimum amount of shared information with obfuscated code and the clear code. It is important to study the relation between the security parameter and the original (clear) code, as it may influence the outcome the obfuscation process. In the following theorem, we use a simple way to check the independence between λ and P on the sub-program level.

Theorem 4.13. *Let Q be a set of obfuscated sub-programs, λ a set of security parameters (obfuscation keys) such that $\lambda = \{\kappa_1, \ldots, \kappa_n\}$, and P a set of clear sub-programs, such that each sub-program q of Q has length at most n, and is the obfuscation of a corresponding block in P: there are $p_i \in P, \kappa_i \in \lambda$ such that q is an obfuscated program of p_i using security parameter κ_i. If $K(\kappa_i, p_i) \geq K(p_i) + K(\kappa_i) - \alpha$, for some $\alpha \in \mathbb{N}$, then (up to a logarithmic term):*

$$I_K(\kappa_i; p_i) \leq \alpha$$

Proof.

$$\begin{aligned}
K(p_i, \kappa_i) &\leq K(\kappa_i \,|\, p_i) + K(p_i) + O(\log n) && \text{by Theorem 2.36} \\
K(p_i) + K(\kappa_i) - \alpha &\leq K(\kappa_i \,|\, p_i) + K(p_i) + O(\log n) && \text{by assumption} \\
K(\kappa_i) - \alpha &\leq K(\kappa_i \,|\, p_i) + O(\log n) \\
K(\kappa_i) - K(\kappa_i \,|\, p_i) &\leq \alpha + O(\log n) \\
I_K(\kappa_i, p_i) &\leq \alpha + O(\log n) && \text{by Definition 2.35}
\end{aligned}$$

□

The following two theorems address the security of two different forms of code obfuscation: obfuscation-as-encoding and obfuscation-as-hiding. In the obfuscation-as-encoding technique, the original program is transformed in such a way it changes the structure of original code, but preserving the functionality, for example Data transformation techniques such as *Array Splitting,*

Splitting Variables, Restructure Arrays and Merge Scalar Variables (see Section 3.4.3). The encoding process is considered as the security parameter that dictates how the obfuscation should be performed and where it should take place.

Encoding differs from encryption; if somebody knows the encoding process, then the original code can be recovered. In encoding obfuscation, the clear program is not presented in the obfuscated code; what still exists, but is hidden, is the encoding process. Reversing the encoded program (obfuscated) requires finding and understanding the encoding process. For instance, we use a simple encoding in Fig. 4.1: `x=x+i` is encoded as `x=x+4*i;x=x-2*i;x=x-i`; the encoding process converts `i` to `4*i;-2*i;-i`, to figure out `x=x+i` , we have to find and combine `4*i;-2*i;-i`, which is the security parameter in this case.

The next theorem addresses the security of obfuscation code when applying an encoding to P (as a set of sub-programs); here the encoding process is presented as a part of security parameter. We aim to check how much the obfuscated sub-program leaks about the encoding process (the security parameter). It turns out that the security of obfuscation-as-encoding depends on the randomness deficiency (see Definition 2.34) of the relevant security parameter of the obfuscated sub-program, as the next theorem shows.

Theorem 4.14 (Encoding)**.** *Let Q be a collection of obfuscated sub-programs q_i of length at most n using $\kappa_i \in \lambda$, a set of security parameters such that $\lambda = \{\kappa_1, \ldots, \kappa_n\}$. Then,*

$$I_K(\kappa_i; q_i) \leq \delta_{\kappa_i} - O(1),$$

for $\delta_{\kappa_i} = \delta(\kappa_i \,|\, Q)$.

Proof. Since Q is a collection of sub-programs, we can assume that it contains all the information in q_i as well as that of all other sub-programs. Then, $K(\kappa_i|Q) = K(\kappa_i|\langle q_1, \ldots, q_i, \ldots, q_n\rangle) \leq K(\kappa_i|q_i)$, as adding more inputs reduces the overall complexity, and so

$$\begin{aligned} I_K(\kappa_i; q_i) &= K(\kappa_i) - K(\kappa_i \,|\, q_i) && \text{by Definition 2.35} \\ &\leq K(\kappa_i) - K(\kappa_i \,|\, Q) \\ &\leq K(\kappa_i) - (\log \#Q - \delta_{\kappa_i}) && \text{by Definition 2.34} \end{aligned}$$

```
int n, i=0; x=0; y=0;
while(i<n)
  i=i+1;
if(7*y*y-1==x*x) //false
  y=x*i;
else
  x=x+i;
```

Figure 4.3: Obfuscating `x=x+i` expression using opaque predicate with no encoding

Because each sub-program in Q has length at most n, then Q can contain at most 2^n distinct sub-programs. Assuming that each appears at most a constant number of times, we have that $\#Q = O(2^n)$ and $\log \#Q = n + O(1)$. Then, by Definition 2.34

$$I_K(\kappa_i; Q_i) \leq n - n + \delta_{\kappa_i} - O(1) = \delta_{\kappa_i} - O(1)$$

□

In obfuscation-as-hiding techniques, the original sub-program still exists in the obfuscated program (set of obfuscated sub-programs), the security of such techniques depends on the degree of hiding in the set of obfuscated sub-programs. An example is the Control-Flow obfuscations such as *Insert Dead basic-blocks, Loop Unrolling, Opaque Predicate and Flatten Control-Flow* (see Section 3.4.2). Normally these techniques are combined and used with encoding obfuscation techniques, in order to make the code more resilient to reverse engineering techniques. For code obfuscation, an opaque predicate is used as a guard predicate that cannot statically be computed without running the code; however the original code still also exists in the obfuscated code, but protected by the predicate. In Fig. 4.1 we used opaque predicates with a simple data encoding technique. Consider the obfuscated code in Fig. 4.3 of the original code in Fig. 4.1-(a), where the encoding has been removed. Obviously, `x=x+i` is still in the code, but is hidden under the protection of opaque predicate. The security of obfuscation-as-hiding depends on the randomness deficiency of the clear sub-program that is obfuscated, and still (hidden) exists in the set obfuscated sub-programs. The next theorem states this case.

Theorem 4.15 (Hiding)**.** *Let Q be a collection of obfuscated sub-programs q_i hiding original*

sub-programs p_i, each of length at most n. Then,

$$I_K(p_i; q_i) \leq \delta_{p_i} - O(1),$$

for $\delta_{p_i} = \delta(p_i \mid Q)$.

Proof. The proof is very similar to Theorem 4.14. The block p_i is hidden in Q but in its original form, due to the obfuscation. Since Q is a collection of sub-programs, we can assume that it contains all the information in q_i as well as that of all other sub-programs. Then, $K(p_i \mid Q) = K(p_i \mid \langle q_1, \ldots, q_i, \ldots, q_n \rangle) \leq K(p_i \mid q_i)$

$$\begin{aligned} I_K(p_i; q_i) &= K(p_i) - K(p_i \mid q_i) \\ &\leq K(p_i) - K(p_i \mid Q) \\ &\leq K(p_i) - (\log \#Q - \delta_{p_i}) \text{ by Definition 2.34} \end{aligned}$$

Similarly to the proof of Theorem 4.14, $\#Q = O(2^n)$ and $\log \#Q = n + O(1)$. Then,

$$I_K(p_i; q_i) \leq n - n + \delta_{p_i} - O(1) = \delta_{p_i} - O(1)$$

□

4.4 Combining Obfuscation Transformation Functions

In this section we investigate the effect of combining different obfuscation techniques on a program. Studying the effect of multiple obfuscation techniques is important. The general belief is: if we apply more obfuscation then the obfuscated code becomes more complicated, complex and difficult to analyse, especially for a malware writer and virtual machine obfuscation [FWWH11]. However, that is assumed to hold with no formal or empirical evidence for support.

Investigating the effect of applying many obfuscation techniques is essential when using dynamic obfuscation. Dynamic obfuscation is very similar to metamorphic malware and self-modifying code; the code gets obfuscated during runtime, and each instance or version may be obfuscated using different obfuscation techniques. Therefore, it is desirable to reason about the security of combining many obfuscation transformations i.e. for such constructions to be secure, the original

obfuscation has to be secure too under composition.

Let $\mathcal{O}^n(P,\lambda)$ denote an obfuscation function that is applied recursively n times to P such that: $O^n(P,\lambda) = \mathcal{O}^1(\mathcal{O}^{n-1}(P,\lambda),\lambda)$, where $\mathcal{O}^1(P,\lambda) = \mathcal{O}(P,\lambda)$. The next theorem states the effect of multiple obfuscation on an obfuscated code using the same obfuscated technique and the same security parameter.

Theorem 4.16.

$$K(\mathcal{O}^n(P,\lambda)) \leq K(\mathcal{O}(P,\lambda)) + \sum_{i=1}^{n} K(\lambda \,|\, \mathcal{O}^{i-1}(P,\lambda)) + O(1).$$

Proof. We proof this theorem using induction:

- **Base case**: we need to prove $K(\mathcal{O}(\mathcal{O}(P,\lambda),\lambda)) \leq K(\mathcal{O}(P,\lambda)) + K(\lambda \,|\, \mathcal{O}(P,\lambda)) + O(1)$: Applying Theorem 2.28 and using the chain rule of Theorem 2.36:

$$K(\mathcal{O}(\mathcal{O}(P,\lambda),\lambda)) \leq K(\mathcal{O}(P,\lambda),\lambda) + O(1) = K(\mathcal{O}(P,\lambda)) + K(\lambda \,|\, \mathcal{O}(P,\lambda)) + O(1)$$

- **Inductive case:** let $\forall i \in \mathbb{N}. K(\mathcal{O}^i(P,\lambda)) \leq K(\mathcal{O}(P,\lambda)) + K(\lambda \,|\, \mathcal{O}(P,\lambda)) + ... + K(\lambda \,|\, \mathcal{O}^{i-1}(P,\lambda)) + O(1)$ holds, then we need to prove:

$$\forall i \in \mathbb{N}. K(\mathcal{O}^{i+1}(P,\lambda)) \leq K(\mathcal{O}(P,\lambda)) + K(\lambda \,|\, \mathcal{O}(P,\lambda)) + ... + K(\lambda \,|\, \mathcal{O}^{i}(P,\lambda)) + O(1)$$

$$K(\mathcal{O}^{i+1}(P,\lambda)) = K(\mathcal{O}(\mathcal{O}^i(P,\lambda),\lambda)) \leq K(\mathcal{O}^i(P,\lambda),\lambda) + O(1)$$

By chain rule of Theorem 2.36:

$\leq K(\mathcal{O}^i(P,\lambda)) + K(\lambda \,|\, \mathcal{O}^i(P,\lambda))$

By Theorem 2.28 and the inductive case:

$\leq K(\mathcal{O}(P,\lambda)) + K(\lambda \,|\, \mathcal{O}(P,\lambda)) + ... + K(\lambda \,|\, \mathcal{O}^{i-1}(P,\lambda)) +$
$K(\lambda \,|\, \mathcal{O}^i(P,\lambda)) + O(1)$

Therefore, $\forall n \in \mathbb{N}.\ K(\mathcal{O}^n(P,\lambda)) \leq K(\mathcal{O}(P,\lambda)) + K(\lambda \,|\, \mathcal{O}(P,\lambda)) + ... + K(\lambda \,|\, \mathcal{O}^{n-1}(P,\lambda)) + O(1)$ holds.

□

We can see the maximum effect of applying the same obfuscation technique many times on obfuscated code complexity. The obfuscated program is maximized when each instance of the obfuscated code (as a result of multiple obfuscation) has the minimum shared information with the security parameters i.e. $K(\lambda \mid \mathcal{O}^n(P,\lambda)) = K(\lambda)$, the conditional input $\mathcal{O}^n(P,\lambda)$ has no effect on computing λ; hence, the shortest program to compute λ is no less than λ itself. Furthermore, the maximum complexity of multiple obfuscations can be increased linearly in the best possible scenarios i.e. $O(n)$.

Theorem 4.17. $K(\mathcal{O}^n(P,\lambda)) \leq K(\mathcal{O}(P,\lambda)) + (n-1)K(\lambda) + O(1)$.

Proof. According to Theorem 4.16,

$$K(\mathcal{O}^n(P,\lambda)) \leq K(\mathcal{O}(P,\lambda)) + K(\lambda \mid \mathcal{O}(P,\lambda)) + ... + K(\lambda \mid \mathcal{O}^{n-1}(P,\lambda)) + O(1),$$

using Theorem 2.27, then:

$$K(\mathcal{O}^n(P,\lambda)) \leq K(\mathcal{O}(P,\lambda)) + \overbrace{K(\lambda) + ... + K(\lambda)}^{n-1} + O(1)$$

□

This theorem computes the maximum achievable Kolmogorov complexity by multiple obfuscation repeating the same technique, but it does not guarantee any lower bound. We illustrate this in the following example:

Example 4.18. *Consider again the obfuscation of Example 4.4, $\mathcal{O}(P,\lambda) = P \parallel \lambda$. If we apply this obfuscation n times times to P using the same λ, we get:*

$$K(\mathcal{O}^n(P,\lambda)) \leq K(P \parallel \lambda) + \sum_{i=1}^{n-1} K(\lambda \mid \langle P \overbrace{\parallel \lambda \parallel ... \parallel \lambda}^{i} \rangle) + O(1).$$

Let n denote the length of P. Then, $K(\lambda \mid \langle P \overbrace{\parallel \lambda \parallel ... \parallel \lambda}^{i} \rangle) \leq \log n$ for any $i \geq 1$: since λ appears in the strings $P \overbrace{\parallel \lambda \parallel ... \parallel \lambda}^{i}$, all we require to produce λ is to identify where it starts, which can be given by the length of P. This takes $O(\log n)$ bits to describe.

Then, $K(\mathcal{O}^n(P,\lambda)) \leq K(P \parallel \lambda) + O(\log n)$, and we can see that applying this obfuscation to P more than one time is irrelevant.

Now, let $\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n)$ stand for an obfuscation process that successively applies n different obfuscation functions to P, using different security parameters such that:

$$\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n) = \mathcal{O}_{\vec{t}_{n-1}.t_1}(P, \vec{\lambda}_{n-1}.\lambda_1) = \mathcal{O}_{\vec{t}_{n-1}}(\mathcal{O}_{t_1}(P, \lambda_1), \vec{\lambda}_{n-1})$$

where $\mathcal{O}_{t_i}$ represents an obfuscation using transformation algorithm t_i, and $\vec{\lambda}_n$ is a vector of different security parameters. The next theorem examines the effect of this successive obfuscation on complexity.

Theorem 4.19. $K(\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n)) \leq K(\mathcal{O}_{t_1}(P, \lambda_1)) + \sum_{i=2}^{n} K(\lambda_{i-1} \,|\, \mathcal{O}_{\vec{t}_i}(P, \vec{\lambda}_i)) + O(1)$

Proof. The proof is very similar to that of Theorem 4.16. □

From the above results, we can see the complexity of an obfuscated program as a result of applying different obfuscation transformations greatly depends on the effect of a new obfuscation function on the previous obfuscated instance, in particular the security parameter. The maximum obfuscation level is achieved when each obfuscation parameter is independent of the corresponding obfuscated instance, i.e. $K(\lambda_i) = K(\lambda_i \,|\, \mathcal{O}_{\vec{t}_{i-1}}(P, \vec{\lambda}_{i-1}))$. Moreover, if each obfuscation function in $\Psi = \{\mathcal{O}_{t_j} \mid j \in \mathbb{N}\}$ is applied independently to different modules or blocks of the original program $P = \{p_i \mid i \in \mathbb{N}\}$, i.e. each obfuscation function $\mathcal{O}_{t_i} \in \Psi$ is mapped (obfuscated) to one module in $p_i \in P$, then we localise a unique obfuscation transformation function to a particular code block or module. In the following Lemma, we prove the optimality of the security parameters in the obfuscation process, which involves applying different obfuscation techniques using different security parameters, in a similar way to Lemma 4.11.

Lemma 4.20. *Given $\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n)$ an obfuscation function that is applied recursively n times to P, using a set of different obfuscation functions: $\vec{\mathcal{O}} = \{\mathcal{O}_{t_1}, \ldots, \mathcal{O}_{t_n}\}$ and a set of security parameters: $\vec{\lambda} = \{\lambda_1, \ldots, \lambda_n\}$ such that: $\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n) = \mathcal{O}_{\vec{t}_{n-1}.t_1}(P, \vec{\lambda}_{n-1}.\lambda_1) = \mathcal{O}_{\vec{t}_{n-1}}(\mathcal{O}_{t_1}(P, \lambda_1), \vec{\lambda}_{n-1})$. Then, $K(\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n) \,|\, P) \leq K(\vec{\lambda}_n \,|\, P)$ up to a logarithmic term.*

Proof. Construct a function F such that $F(\vec{\mathcal{O}}, \vec{\lambda}, .) = \mathcal{O}_{\vec{t}_n}(., \vec{\lambda}_n)$, let Q_K be the program that implements it. Similarly to the proof of Lemma 4.11, $U(Q_K, P) = F(< \vec{\mathcal{O}}_{\vec{t}_n}, \vec{\lambda}_n >, P)$ and so $|Q_K| \geq K(F(< \vec{\mathcal{O}}_{\vec{t}_n}, \vec{\lambda}_n >, P) \,|\, P)$. To describe Q_K we need to describe every single instance of $\vec{\lambda}_n$ and invoke a set of instruction to determine which obfuscation function to use and in which

order. Since we have P, we can use it to describe each instance of $\vec{\lambda}_n$ with a short description. We still have to describe the obfuscation functions. Assume $\vec{\mathcal{O}}$ is a recursive enumerable set, then by Lemma 2.33 it is sufficient to pin-point the index of obfuscation function $\mathcal{O}$ in a set $\vec{\mathcal{O}}$ so that

$$K(F(<\vec{\mathcal{O}}, \vec{\lambda}>, P) \mid P) \leq K(\vec{\lambda}_n | P) + \log \#\vec{\mathcal{O}}.$$

Therefore, $K(\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n) \mid P) \leq K(\vec{\lambda}_n \mid P) + O(\log n)$. □

Having provided the optimality in cases of multiple obfuscation, we turn to provide security results in the following theorem.

Theorem 4.21. *Consider a set of obfuscation functions $\vec{\mathcal{O}} = \{\mathcal{O}_{t_1}, \ldots, \mathcal{O}_{t_n}\}$ and a set of security parameters: $\vec{\lambda} = \{\lambda_1, \ldots, \lambda_n\}$ as before. Let P represent a program of length m. Assume that the keys (security parameters) are random, $\forall \lambda_i \in \vec{\lambda}, \exists \alpha_i . K(\lambda_i) \geq m - \alpha_i$; the segments are mutually independent even in the presence of P, $|K(\vec{\lambda}_n) - \sum_{i=1}^{n} K(\lambda_i | P)| \leq O(\log m)$; independent from P, $\forall \lambda_i \in \vec{\lambda}, \exists \beta_i . K(\lambda_i \mid P) \geq K(\lambda_i) - \beta_i$; and satisfying $K(\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n) \mid P) \geq K(\vec{\lambda}_n \mid P) - O(\log m)$. Then the obfuscated code $\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n)$ is $\sum_{i=1}^{n}\{\alpha_i + \beta_i\}$-secure up to a logarithmic term on the size of P.*

Proof.

$$\begin{aligned}
I_K(P; \mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n)) &= K(\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n)) - K(\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n) \mid P) \\
&\leq K(\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n)) - K(\vec{\lambda}_n \mid P) + O(\log m) && \text{by Lemma 4.20} \\
&\text{by the assumption on randomness and independence of } \vec{\lambda} \\
&\leq K(\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n)) - \sum_{i=1}^{n} K(\lambda_i \mid P) + O(\log m) \\
&\text{By the assumption on } K(\lambda_i \mid P) \text{ and } K(\lambda_i): \\
&\leq m - \sum_{i=1}^{n}\{K(\lambda_i) - \beta_i\} + O(\log m) \\
&\leq m - \sum_{i=1}^{n}\{m - \alpha_i - \beta_i\} + O(\log m) \\
&\leq \sum_{i=1}^{n}\{\alpha_i + \beta_i\} + O(\log m)
\end{aligned}$$

□

4.5 Summary

In this chapter we provided a theoretical investigation of code obfuscation security. We defined code obfuscation using Kolmogorov complexity and algorithmic mutual information. Our definition allows for a small amount of secret information to be revealed to an adversary, and it gives an intuitive guarantee about the security conditions that have to be met for secure obfuscation. We argued our definition is more lenient than the virtual black-box model of Barak et al. and that for that reason the impossibility result does not apply. In contrast, we showed that under reasonable assumptions we can have secure obfuscation according to our definition.

5 Modeling Adversaries using Kolmogorov Complexity

In this chapter we propose a generic model for a code obfuscation adversary based on Algorithmic Information Theory and Kolmogorov complexity. We provide a formal grounding, a general description and a new definition of code obfuscation adversary including the adversary's objectives. We explore its potential and the attacking process.

5.1 Introduction

In all different aspects of information security, the adversary model represents the corner stone for understanding and defining security of any system. In the previous chapter, the adversary model was implicit in our definition of security: the adversary knows the obfuscated code and its objective is to produce the original code. However, the definition we have provided, which is based on Algorithmic Information Theory (as with Classical Information Theory), only accounts for an adversary with unbounded computational power. There are no limits being imposed on the adversary's computational power, which evidently presents a very powerful attacker. As with cryptography, this type of adversary is impractical, and there is almost no defence mechanism that can resist such an attacker.

In the practical cases of attacking code obfuscation, the adversary is equipped with a limited number of reverse engineering tools at her/his disposal, besides the bounded time. It is very beneficial to consider such cases, although the context in which the attacker is operating is fully open. This type of attack is called *Man-at-End (MATE)*, where the adversary has full access to the computational resources, including physical access to the device. S/he can execute, tamper, modify and inspect hardware and software freely without any restriction. Our definition for obfuscation security, which is based on algorithmic mutual information, implicitly models such a

powerful adversary through the conditional Kolmorgorov complexity. However, its very important to explicitly define such adversary by providing a formal attack model, which can demonstrate her/his capabilities, goals, the attack process, and its limitation; hence, we can provide a rigorous meaning for code obfuscation security. Furthermore, the adversary model has to account for the situations where the adversary has a limited resources to use in the attack.

To the best of our knowledge, this is the first attempt to provide a formal and mathematical model for a code obfuscation adversary; we are not aware of any formal model for a generic adversary model for code obfuscation, apart from the traditional static and dynamic program analysis tools.

The remainder of this chapter is organised as follows. Section 5.2 describes the obfuscation adversary model. In Section 5.3 we define a legitimate adversary. Section 5.4 investigates the security of an obfuscated program, using multiple obfuscation transformations, against an adversary.

5.2 Obfuscation Adversary Model

5.2.1 Adversary Capabilities

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 thereof, without the assistance of any automated tools. The difficulty of the adversary'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 adversary 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 adversary with access to a complete range of analysis tools. S/He can execute the obfuscated code (which we refer to as a *challenge*) as many times as required; s/he can run any program on the challenge (that is, it can execute any computable function of the obfuscated code) to obtain static analysis information on that code or, on the other hand, to produce modifications and variations of the challenge code, which s/he can subsequently run as many times as needed. The adversary is not restricted in how many times s/he runs these functions or modified obfuscated programs nor in what order. Ultimately, with all the information gathered from this process, the adversary will attempt to produce a deobfuscated version of the program, that is, well-structured code which is very similar to the original program and has less noise.

This model equally captures automated reverse-engineering analysis techniques such as static program analysis and dynamic program analysis (see Section 3.2). A distinction commonly made in the literature is whether the attacker is *passive* or *active*. The first kind of adversary is limited to analysing the source code of the challenge, but cannot run it nor modify it. The second adversary can execute the challenge code and the modified versions thereof. We model these two kinds of adversary in the choice of computable functions that are available to the adversary.

5.2.2 Adversary Goals

On the other hand, there is also some lee-way in the adversary's goal. The most strict victory condition is to produce the original code. More relaxed conditions would allow the adversary to win if it could produce code that was close enough to the original source and had the same functionality. In our scenario, the adversary already knows the functionality to a large degree. If s/he did not, an adequate victory condition would be simply to produce as simple as possible an equivalent version of the obfuscated code, which would mean the adversary had understood the functionality and found a more compact implementation for it. But this does not mean the adversary's implementation is better than the original in some practical terms, for example, more efficient. Our aim is to represent a situation where the obfuscated implementation of a specific functionality holds some value for the attacker; although the latter might know the full functionality, the way in which this is implemented is not known, and might be better in some practical terms than all the implementations the adversary currently knows. Given the above, it seems to us that the adversary's goal must be to recover code as close as possible to the original, and definitely with similar practical properties. This last requirement is difficult to formalise, because the possible criteria are so many, so it is easier to settle with the strict condition.

5.2.3 Adversary Definition

In this section we provide a definition for a generic attacker for code obfuscation and software protection based on Kolmogorov complexity. We start by providing a new definition for an adversary (deobfuscator), in a similar way to the code obfuscation definition (see Definition 4.2).

Intuitively, an adversary with deobfuscation capabilities can be considered as a transformation of an obscure program into a more intelligible program, which is functionally equivalent to the original program [BF07]. Kolmogorov complexity can be used to define intelligibility as it measures the

level of regularity in programs.

Definition 5.1. *(Deobfuscation Adversary) A* deobfuscation adversary $\mathcal{A} : \mathcal{P}' \rightarrow \mathcal{P}$ *is a function that maps an obfuscated program to an approximate version of the original program. The deobfuscation outcome,* $D(P) = \mathcal{A}(\mathcal{O}(P, \lambda))$*, has the following properties:*

- ***Functionality****: for an obfuscated program* $\mathcal{O}(P, \lambda)$ *and its original code* P*, applying the deobfuscation gives a program with the same functionality, i.e. :* $[\![P]\!] = [\![D(P)]\!]$.[1]
- ***Polynomial Slowdown****: for any obfuscated program* $\mathcal{O}(P, \lambda)$*, the size and running time of* $D(P)$ *are at most polynomially larger than the size and running time of* P *i.e.:*

$$|D(P)| \leq p(|P|).$$

- ***Proximity****: The deobfuscation is* δ*-close if :*

$$|K(P) - K(D(P))| \leq \delta K(P)$$

  for $0 \leq \delta \leq 1$

- ***Intelligibility****: Deobfuscation is c-intelligible if:*

$$\exists c > 0 \; . \; K(\mathcal{O}(P, \lambda)) \geq cK(D(P))$$

  Where $|P|$ *and* $|\mathcal{O}(P, \lambda)|$ *are of* $O(n)$.

The proximity property explains the situation where the adversary is trying to recover a program that is as close as possible to the original program complexity; alternatively it can be stated in a less compact form as $|K(P) - K(D(P))| \leq \delta K(P)$. Additionally, the result code should not be significantly more complex than the true original code, which means they must have also similar Kolmogorov complexity. This prevents cases where the adversary has not sufficiently reduced the complexity of the code, and has returned something that is probably still rather obfuscated. This does not mean the original and the candidate programs have high mutual information and can be easily derived from each other, it only means that they have similar amounts of information.

[1] If this cannot be checked, e.g due to a large function domain, the adversary must provide a formal proof that the functionality is the same.

The intelligibility property sounds intuitive here, because most of deobfuscation strategies by passive and active adversaries, whether static or dynamic, are aiming at producing less complex code, in order to make it as simple as possible for understanding. However, reducing the complexity of obfuscated code does not provide a guarantee about adversaries' abilities. Take for example the obfuscated code in Fig. 4.2, consider an active adversary $\mathcal{A}$ who constructs, successfully, an intelligible version $D(P)$ of $\mathcal{O}(P,\lambda)$, by replacing all these variables by meaningful ones. However, by using the same reasoning of Proposition 4.9, $D(P)$ can be of higher complexity compared to $\mathcal{O}(P,\lambda)$, as there is more information to be described in $D(P)$. Therefore, c-intelligibility is not satisfied by this attack, although $\mathcal{A}$ managed to produce an intelligible program from $\mathcal{O}(P,\lambda)$.

This leads us to consider a further definition to compute the adversary's outcome, where the adversary's success is captured by the notion of algorithmic mutual information. We consider the revealed information about the original code as the adversary's *advantage* in attacking the obfuscated code. It can be stated formally as:

Definition 5.2. *Given an obfuscated code $\mathcal{O}(P,\lambda)$ and an active adversary $\mathcal{A}$; the adversary's advantage is defined as:*

$$Adv_{\mathcal{A}}^{P,\mathcal{O}(P,\lambda)} = I_K(P;\mathcal{O}(P,\lambda))$$

If $\mathcal{A}$ has α, which contains some knowledge about P, then:

$$Adv_{\mathcal{A}}^{P,\mathcal{O}(P,\lambda)}(\alpha) = I_K(P;\mathcal{O}(P,\lambda)\,|\,\alpha)$$

The above definition is very similar to Definition 4.5, the adversary is implicitly considered using conditional Kolmogorov complexity, with one difference: it accounts for an adversary $\mathcal{A}$ with additional knowledge about P. The notation $Adv_{\mathcal{A}}^{P,\mathcal{O}(P,\lambda)}$ does not indicate a different mutual information nor a restrictive version. It explicitly states the existence of an adversary $\mathcal{A}$ that is trying to describe P given the knowledge of the obfuscated version $\mathcal{O}(P,\lambda)$.

Having a high advantage $Adv^{P,\mathcal{O}(P,\lambda)}$ means the obfuscated code is less secure and shows a high capability for that adversary, and vise versa. However, this definition as it is cannot be used to measure the adversary's success at attacking the obfuscated code; if the advantage is equal to γ, then the adversary reveals no more than what the defender wants the adversary to learn i.e. the allowed information to be leaked, γ (see Definition 4.5).

The next proposition provides an example of how to compute an adversary's advantage that

attacks an obfuscated program. It shows that the adversary's advantage coincides with the amount of retrieved information from obfuscated code with only up to logarithmic precision.

Proposition 5.3. *Consider a clear program P of length n with $K(P) \geq n - \alpha$, for some $\alpha > 0$, and $\mathcal{A}$ an adversary that extracts at least $m \leq n$* consecutive *bits of P from $\mathcal{O}(P, \lambda)$ then:*

1. $K(P \mid \mathcal{O}(P, \lambda)) \leq n - m + O(\log n)$.
2. $Adv_{\mathcal{A}}^{P,\mathcal{O}(P,\lambda)} \geq m + \alpha + O(\log n)$

Proof. We prove this proposition by building the following algorithm:

Algorithm:

- Run $\mathcal{A}(\mathcal{O}(P, \lambda))$; we obtain $m \leq n$ bits of P. Denote this by ω.
- Now, run a program β such that $P = (s_1, s_2) = \beta(\omega)$ which computes two blocks of bits: those that come before and those that come after ω in P.

  To produce P, β needs at most to produce the pair of two strings (s_1, s_2) with combined length $n - m$. To describe the pair, we need at most $O(\log n)$ bits saying where to divide s_1 from s_2. Thus, $K(\beta) \leq |\beta| \leq n - m + O(\log n)$.

By construction, $K(P \mid \mathcal{O}(P, \lambda)) + O(1) \leq K(\beta) + O(1) \leq n - m + O(\log n)$. Using the assumption about $K(P)$, it is straight forward to compute $I_K(P, \mathcal{O}(P, \lambda)) \geq m + \alpha + O(\log n)$. □

An adversary can fully obtain P, with only a logarithmic error, if s/he knows λ, the security parameter (obfuscation key) that is used to obfuscate.

Lemma 5.4. *Given an obfuscated program $\mathcal{O}(P, \lambda)$, for an adversary $\mathcal{A}$ who knows λ, the security parameter.*

1. $K(P \mid \mathcal{O}(P, \lambda), \lambda) = O(\log n)$.
2. $Adv_{\mathcal{A}}^{P,\mathcal{O}(P,\lambda)}(\lambda) = K(P \mid \lambda) - O(\log n)$.

where n is the length of $P, \mathcal{A}$ and λ.

Proof. By secrecy property of Definition 4.2, $\lambda \in \mathcal{L}$ contains all that $\mathcal{A}$ needs to obtain P from $\mathcal{O}(P, \lambda)$. The shortest program for a Universal Turing machine U that describes P given $\mathcal{A}, \mathcal{O}(P, \lambda)$ and λ, is logarithmic; as it is sufficient for U to describe P from $\mathcal{A}, \mathcal{O}(P, \lambda)$ and λ

using just $O(\log n)$, which is the overhead cost required by U to combine $\mathcal{A}, \mathcal{O}(P,\lambda)$ and λ and to locate them on U tape. The advantage of $\mathcal{A}$ given λ is obtained as follows:

$$\begin{aligned} Adv_{\mathcal{A}}^{P,\mathcal{O}(P,\lambda)}(\lambda) &= I_K(P;\mathcal{O}(P,\lambda)\,|\,\lambda) && \text{by Definition 5.2} \\ &= K(P\,|\,\lambda) - K(P\,|\,\mathcal{O}(P,\lambda),\lambda) \\ &= K(P\,|\,\lambda) - O(\log n) \end{aligned}$$

□

These results are not surprising. Intuitively, an adversary can easily recover the original code from the obfuscated version once the security parameter that is used for obfuscation is known. This is an extreme case where the adversary has a possession of the security parameter, which it can be used to fully deobfuscate the obfuscated code.

If the adversary has a prior information about the original code or the security parameters used in the obfuscation process, then we can use the result of the next theorem to estimate its outcome, based on the complexity of the clear code and the security parameter, which shows the amount of resilience of code obfuscation against the adversary.

Theorem 5.5. *For an adversary $\mathcal{A}$:*

$$\begin{aligned} Adv_{\mathcal{A}}^{P,\mathcal{O}(P,\lambda)} &\geq K(P) - \mathit{min}\{K(P), K(\lambda)\} - O(\log n) \\ Adv_{\mathcal{A}}^{P,\mathcal{O}(P,\lambda)}(\alpha) &\geq K(P\,|\,\alpha) - \mathit{min}\{K(\lambda\,|\,\alpha), K(P\,|\,\alpha)\} - O(\log n) \end{aligned}$$

where n is the length of $P, \mathcal{A}$, λ and α.

Proof. $K(P\,|\,\mathcal{O}(P,\lambda)) \leq K(P)$ by Theorem 2.27, and $K(P\,|\,\mathcal{O}(P,\lambda))$ cannot exceed $K(\lambda)$ i.e. $K(P\,|\,\mathcal{O}(P,\lambda)) \leq K(\lambda)$, as P can be recovered using $\mathcal{O}(P,\lambda)$ and λ by Lemma 5.4, and Theorem 2.30. Therefore $K(P\,|\,\mathcal{O}(P,\lambda)) \leq \min\{K(P), K(\lambda)\} + O(\log n)$, since Kolmogorov complexity require the shortest possible program.

The proof to the conditional part, is very similar to the above proof with a minor change, it just requires adding α to both sides of inequality i.e. having $K(P\,|\,\mathcal{O}(P,\lambda),\alpha)$. □

The previous theorem sets the lower bound on adversary's advantage. In Lemma 5.4 we provide an extreme case where the adversary has the security parameter, which is used to fully deobfuscate

the protected code. Intuitively, the adversary's advantage, in this case, is the best result that the adversary can achieve. The next theorem states these intuitions formally, by providing the upper bound on the adversary capabilities.

Theorem 5.6. *For an adversary $\mathcal{A}$:*

$$Adv_{\mathcal{A}}^{P,\mathcal{O}(P,\lambda)} \leq K(P) - O(\log n)$$

where n is the length of $P, \mathcal{A}$ and λ.

Proof. To prove this inequality, we follow the same methodology that was used to proof Muchnik's Theorem [Muc11], with changes that fit our theory. We construct $\mathcal{O}(P,\lambda) = P \otimes \lambda$ as an obfuscated version of P using λ, where $\otimes$ is an operator used to obfuscate (encode or hide see Section 4.3). Applying the results of Lemma 5.4, we obtain $P = \mathcal{A}(\mathcal{O}(P,\lambda)) \otimes \lambda$, the outcome of attacking $\mathcal{O}(P,\lambda)$ when λ is known to $\mathcal{A}$. Using the conditional description of Theorem 2.30, we can construct λ' such that $K(\lambda' \mid \lambda) = O(\log n)$, here λ' is relatively simple compared to $K(\lambda)$ with only a logarithmic difference. Now, we have the following derivations, with only a logarithmic term in each step:

$$K(P \mid \mathcal{O}(P,\lambda)) = K(\lambda, \mathcal{O}(P,\lambda) \mid \mathcal{O}(P,\lambda)) + O(\log n)$$

by Theorem 2.30

$$= K(\lambda, \lambda', \mathcal{O}(P,\lambda) \mid \mathcal{O}(P,\lambda)) + O(\log n)$$

from λ' and $\mathcal{O}(P,\lambda)$, $\mathcal{A}$ can construct P, according to Lemma 5.4

$$= K(\lambda, P \mid \mathcal{O}(P,\lambda)) + O(\log n)$$

$$= K(\lambda, P, \mathcal{O}(P,\lambda)) - K(\mathcal{O}(P,\lambda)) + O(\log n) \quad \text{by Theorem 2.36}$$

$$\geq K(\lambda, P) - K(\mathcal{O}(P,\lambda)) + O(\log n)$$

$$\geq K(\lambda, P) - K(P,\lambda) + O(\log n) \quad \text{by Theorem 2.28}$$

By symmetry of information, Theorem 2.36, we have

$$\geq O(\log n).$$

Now, it is straightforward to compute the algorithmic mutual information:

$$Adv_{\mathcal{A}}^{P,\mathcal{O}(P,\lambda)} \leq K(P) - O(\log n).$$

□

The above result provides a upper bound on the adversary's power, which shows that the adversary's advantage should not exceed the complexity of the original code. Intuitively, it shows the maximum knowledge (the clear program) that an adversary hopes to extract from the obfuscated program.

5.3 Legitimate Adversary

It is customary to first fix an adversary and then select the challenge at random, so that the adversary cannot be tailored to a specific instance, i.e. the adversary has no information at all about the messages that are used to construct the challenge. This then leads to a probabilistic analysis of the success of the adversary.

Kolmogorov Complexity theory is used to analyse individual instances and avoid the analysis of probabilistic ensembles, and this is the route we follow in our approach. We intend to guarantee that the adversary holds no information about the plain code, but instead of requiring this to be picked at random, we simply state that the *mutual information* between the plain code and the code of a *legitimate adversary* must be very low.

Our model of security is focused on individual instances of an obfuscation process, instead of considering at the same time all possibilities. This makes analysis much easier in a practical setting, since we can choose a particular case of obfuscated code and analyse its security without having to consider all other cases. This also shifts the focus of the security definition in the direction of actual examples of obfuscated code, and away from the obfuscator algorithm *per se*.

Definition 5.7 (Legitimate Adversary)**.** *An adversary $\mathcal{A}$ is said to be legitimate if and only if:* $I_K(P;\mathcal{A}) = O(1)$.

$I_K(P;\mathcal{A}) = O(1)$ excludes the trivial case where a particular adversary already knows the source code or a good deal of it, and could therefore win without having any 'intelligence' to undo

the obfuscation. For a particular obfuscates instance, it is always possible to find a deterministic algorithm that trivially undoes the obfuscation: for example, an algorithm that simply 'knows' the solution and prints it. Since $\mathcal{A}$ can print P even without knowledge of $\mathcal{O}(P, \lambda)$, $K(P \mid \mathcal{A}) = O(1)$ and so $I(P; \mathcal{A}) = K(P) - O(1)$. This case is excluded by this mutual information requirement.

Another trivial algorithm is one that does not have any particular intelligence for a general obfuscated program: it just has a hard-coded list of changes that revert$\mathcal{O}(P, \lambda)$ to P, for example, a list of the steps in obfuscation process detailing the state before and after that step. But then, this means $\mathcal{A}$ already contains in its code parts that are specific to P: some attacker can look into $\mathcal{A}$ and thus reconstruct these parts of P even without knowing $\mathcal{O}(P, \lambda)$. The complexity of these parts is the information contained in $\mathcal{A}$ about P and if P and $\mathcal{O}(P, \lambda)$ are reasonably different, the complexity of this list will be larger than $O(1)$.

This case is subtly different from one where the obfuscation process is so weak; it is possible to write a simple reversal algorithm, say, R, that simply undoes each step of $\mathcal{O}$. In this case, $\mathcal{A}$ would use R as a list of steps to turn $\mathcal{O}(P, \lambda)$ into P, much like in the previous case, but with the difference that these can be applied to any obfuscated instance $\mathcal{O}(P, \lambda)$ of this obfuscator. For example, containing instructions that convert a particular structure into another, or add or remove specific letters to a variable name. In this case, $\mathcal{A}$ knows R, but this does not have any information about a specific P, which makes $\mathcal{A}$ a legal adversary, giving evidence that $\mathcal{O}$ is a bad obfuscator and instances computed from it are weak.

5.4 The Case of Multiple Obfuscation Transformations

So far we have studied the adversary's advantage for a single obfuscation transformation. In this section we investigate the advantage of an adversary attacking an obfuscated program, which is obfuscated by applying rounds of different obfuscation algorithms in an iterative way. The following theorem shows the resilience against an attacker.

Theorem 5.8. *Given $\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n)$ an obfuscation function that is applied recursively n times to P, using a set of different obfuscation functions: $\vec{\mathcal{O}} = \{\mathcal{O}_{t_1}, \ldots, \mathcal{O}_{t_n}\}$ and a set of security parameters: $\vec{\lambda} = \{\lambda_1, \ldots, \lambda_n\}$ such that: $\mathcal{O}_{\vec{t}_n}(P, \vec{\lambda}_n) = \mathcal{O}_{\vec{t}_{n-1}.t_1}(P, \vec{\lambda}_{n-1}.\lambda_1) = \mathcal{O}_{\vec{t}_{n-1}}(\mathcal{O}_{t_1}(P, \lambda_1), \vec{\lambda}_{n-1})$, for*

an adversary $\mathcal{A}$ with knowledge α:

$$Adv_{\mathcal{A}}^{P,\mathcal{O}_{\vec{t}_n}(P,\lambda)}(\alpha) \geq K(P) - min\{K(P\,|\,\alpha), K(\lambda_1\,|\,\alpha), ..., K(\lambda_n\,|\,\alpha)\} - O(\log n)$$

where n is the length of $P, \mathcal{A}(\mathcal{O}(P,\lambda))$ and α.

Proof. We use induction to proof this theorem:

- **Base case**: using Theorem 5.6

$$K(P\,|\,\mathcal{O}_{t_1}(P,\lambda_1),\alpha) \leq \min\{K(P\,|\,\alpha), K(\lambda_1\,|\,\alpha)\} + O(\log n)$$

- **Inductive case**: let

$$\forall i \in \mathbb{N}. K(P\,|\,\mathcal{O}_{\vec{t}_i}(P,\vec{\lambda}_i),\alpha) \leq \min\{K(P\,|\,\alpha), K(\lambda_1\,|\,\alpha), ..., K(\lambda_i\,|\,\alpha)\} + O(\log n)$$

  then we need to prove:

$$\forall i \in \mathbb{N}. K(P\,|\,\mathcal{O}_{\vec{t}_{i+1}}(P,\vec{\lambda}_{i+1}),\alpha) \leq \min\{K(P\,|\,\alpha), K(\lambda_1\,|\,\alpha), ..., K(\lambda_{i+1}\,|\,\alpha)\} + O(\log n)$$

$$K(P\,|\,\mathcal{O}_{\vec{t}_{i+1}}(P,\vec{\lambda}_{i+1}),\alpha) = K(P\,|\,\mathcal{O}_{t_{i+1}}(\mathcal{O}_{\vec{t}_i}(P,\vec{\lambda}_i),\lambda_{i+1}),\alpha)$$

By Theorem 5.6 and inductive case:

$$\leq \min\{K(P\,|\,\alpha), K(\lambda_1\,|\,\alpha), ..., K(\lambda_i\,|\,\alpha), K(\lambda_{i+1}\,|\,\alpha)\} + O(\log n)$$

Then, by Definition 5.2 we compute:

$$Adv_{\mathcal{A}}^{P,\mathcal{O}_{\vec{t}_n}(P,\lambda)}(\alpha) \geq K(P) - \min\{K(P|\alpha), K(\lambda_1|\alpha), ..., K(\lambda_n|\alpha)\} - O(\log n)$$

□

The previous results demonstrate the effect of different obfuscation techniques with different security parameters, applied to clear code. The next theorem shows the adversary's advantage when the same obfuscation technique, with the same security parameter, is applied iteratively to the original code.

Theorem 5.9. *Let $\mathcal{O}^n(P,\lambda))$ denote an obfuscation function that is applied recursively n times to P such that : $O^n(P,\lambda) = \mathcal{O}^1(\mathcal{O}^{n-1}(P,\lambda),\lambda)$, where $\mathcal{O}^1(P,\lambda) = \mathcal{O}(P,\lambda)$, for an adversary $\mathcal{A}$:*

$$Adv_{\mathcal{A}}^{P,\mathcal{O}^n(P,\lambda)} \geq K(P) - min\{K(P), K(\lambda)\} - O(\log n)$$

where n is the length of P, and $\mathcal{A}$.

Proof. Similarly to the proof of Theorem 5.8,

$$Adv_{\mathcal{A}}^{P,\mathcal{O}^n(P,\lambda)} \geq K(P) - \min\{K(P), \overbrace{K(\lambda) + ... + K(\lambda)}^{n}\} - O(\log n)$$

Then, by Definition 5.2 : $Adv_{\mathcal{A}}^{P,\mathcal{O}^n(P,\lambda)} \geq K(P) - \min\{K(P), K(\lambda)\} - O(\log n)$. □

From the above results, we conclude that the resilience of obfuscated code, which is measured by adversary's advantage $Adv^{P,\mathcal{O}^n(P,\lambda)}$, does not improve as the results of multiple obfuscation using the same security parameters, and the same obfuscation technique. On the other hand, the resilience of applying different obfuscation techniques with different security parameters depends on the Kolmogorov complexity of the security parameters used by different obfuscation techniques. If the security parameters have a high level of independence, i.e. have no shared information with the original code, then they guarantee a high level of security.

5.5 Summary

In this chapter we introduced a new generic model for code obfuscation based on algorithmic information theory. We provided a new definition for an obfuscation adversary and provide the security properties that characterise the success conditions for code obfuscation adversary. We investigated the security of an obfuscated program, using multiple obfuscation transformations, against an adversary.

6 A Theoretical Framework to Measure Code Obfuscation

The purpose of this chapter is to introduce quantitative metrics for software obfuscation. We propose a mathematical framework to measure the quality of code obfuscation, by providing metrics based on Kolmogorov complexity, information distance and compression.

6.1 Introduction

Many obfuscation transformation techniques were proposed in the past, which intuitively can make the program difficult to understand and harder to attack; however, there are no practical security metrics, justified theoretically and empirically, to measure the effectiveness of these obfuscation transformation techniques.

Several attempts were made to provide concrete metrics for evaluating obfuscation [CTL97, AMDS+07], using classical complexity measures. However, most of these metrics are still context dependent and differ among development platform, and therefore it is very hard to standardise them. There is a need to evaluate how obfuscating and deobfuscating transformations affect the understanding of the program, and measure the strength of seemingly resilient obfuscating transformations against reverse engineering and program analysis attacks. 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:

1. Can we derive from Algorithmic Information Theory quantitative metrics, with practical relevance, which can be used to measure the protection level in code obfuscation?

2. How to evaluate the usefulness of these metrics?

In this chapter we are aiming to answer these questions. First, we propose a novel metric for code obfuscation that is based on Kolmogorov complexity and compression. Then we apply the Weyuker validation framework [Wey88] to check whether Kolmogorov complexity is theoretically sound as software metric. We also show that software-similarity metrics, such as information distance [LCL+04] that measures the similarity between two blocks of code, can provide a plausible way to reason about the amount of security of code obfuscation.

In order to evaluate the quality of code obfuscation using similarity metrics, we have to capture quantitatively the degree of confusion in code transformations, taking into account the attacker's capabilities. The aim of using information distance is to quantify the amount of obscured code that remains or is lost when the program is debofuscated.

Similarity metrics are used to determine the diversity of the generated code; however, it does not mean anything in terms of security. We have to establish a way to relate the level of confusion in code obfuscation to the degree of difference in infomation distance between obfuscated code and its original code.

We adapt the work of Li et al. [LCL+04] on the notion of information distance, and extend their theoretical work to reason about the quality of code obfuscation by relating information distance to the security in code obfuscation that was established in the previous chapters.

We formalise the notion of *unintelligibility index* (degree of confusion introduced) and normalised Kolmogorov complexity, and show that information distance metric is a suitable measure for code obfuscation security. In particular, we demonstrate that the information distance between a program and its obfuscated version tends to be bigger if the level of confusion used is higher.

Chapter layout: In Section 6.2 we provide an overview of algorithmic information distance. Section 6.3 provides a theoretical metric validation for Kolmogorov complexity based on Weyuker's complexity axioms. In Section 6.4, we propose a quantitative model to measure code obfuscation based on algorithmic information theory, which comprises a set of metrics : unintelligibility index, normalised Kolmogorov complexity and compression, normalised compression distance, and code obfuscation stealth.

6.2 Information Distance

Li et al. [LCL+04] formalised the notion of information distance, a similarity measure, using Kolmogorov complexity and compression. They showed that the information distance satisfies metric axioms [BGM+98] and can be used as a *measure of distance*; their work was used successfully in the context of bioinformatics measuring the similarities among mitochondrial DNA genomes, also in pattern recognition, data mining, phylogeny [CV05], malware clustering and classification [ABCD15].

The informational distance or similarity distance between two binary strings, x and y, is the quantity of information sufficient to translate between x and y, generating either string effectively from the other. Formally, it is defined as follows.

Definition 6.1 (Information Distance [BGM+98])**.** *The information distance $D_K(x, y)$ is defined as the shortest binary program that computes both x from y and y from x, such that*

$$D_K(x, y) = max\{K(x \mid y), K(y \mid x)\}.$$

The maximum distance is the shortest program (shortest length of a binary program) that computes a maximum amount of information to get from x to y and from y to x, i.e. to measure the absolute quantity of the shared information between x and y.

We can define a conditional version of information distance, which intuitively explains the situation where the information distance supplied by extra information that contributes to the translation between two binary strings x and y. This auxiliary information contributes to the information distance by providing extra to compute x from y, and vice versa.

Definition 6.2 (Conditional Information Distance)**.** *The conditional information distance $D_K(x, y \mid z)$ is defined as the shortest binary program that computes both x from y and y from x, conditioned to z, such that*

$$D_K(x, y \mid z) = max\{K(x \mid y, z), K(y \mid x, z)\}$$

The conditional version of information distance computes the information content that goes from x to y and the information from y to x relative to an auxiliary information z.

The relative quantity of shared information between two binary strings x and y is measured by the normalised information distance (*NID*), which was introduced by Li et al. [LCL+04].

Definition 6.3 (Normalised Information Distance [LCL$^+$04])**.** *The normalised information distance NID_K is defined as*

$$NID_K(x,y) = \frac{max\{K(x\,|\,y), K(y\,|\,x)\}}{max\{K(x), K(y)\}}.$$

Dividing by $\max\{K(x), K(y)\}$ normalises the information distance to take a real value between 0 and 1. In the next definition, we extend the normalised information distance and propose a conditional version of it.

Definition 6.4 (Conditional Normalised Information Distance)**.** *The normalised information distance conditioned to string z is defined as*

$$NID_K(x,y\,|\,z) = \frac{max\{K(x\,|\,y,z), K(y\,|\,x,z)\}}{max\{K(x\,|\,z), K(y\,|\,z)\}}.$$

The normalised version of information distance was proved to satisfy the metric axioms in [LCL$^+$04]; however, we need to make sure that the conditional version we introduced above also satisfies the axiom metrics, see Definition 2.42. We proceed by considering the following Lemma, which is similar to the directed triangle inequality in [GTV01], but only for the conditional version.

Lemma 6.5. *(Conditional triangle inequality) Let x, y, z and u be strings.*

$$K(x\,|\,y,z) \leq K(x,u\,|\,y,z) \leq K(x\,|\,u,z) + K(u\,|\,y,z)$$

up to a logarithmic term.

Proof. Let p be a minimal program that produces x given u, z, and q a minimal program that produces u given y, z. That is, $|p| = K(x\,|\,u,z)$ and $|q| = K(u\,|\,y,z)$.

Then, we can produce a program r that takes inputs y, z and computes

$$\langle x,u\rangle = \langle p(q(y,z),z), q(y,z)\rangle,$$

using programs p, q. A description of r includes a description of p and q plus some negligible code to sequence them (a logarithmic term), and so $K(x,u\,|\,y,z) \leq K(x\,|\,u,z) + K(u\,|\,y,z)$.

Finally, by basic properties of Kolmogorov complexity, $K(x\,|\,y,z) \leq K(x,u\,|\,y,z)$. □

Lemma 6.6. *For string x, y and z, if $K(x\,|\,z) \geq K(y\,|\,z)$, then $K(x\,|\,y,z) \geq K(y\,|\,x,z)$, up to an logarithmic term.*

Proof. By Definition 2.37 and symmetry of mutual information (Theorem 2.36), we have, up to a logarithmic term:

$$I_K(x,y\,|\,z) = K(y\,|\,z) - K(y\,|\,x,z) = K(x\,|\,z) - K(x\,|\,y,z).$$

Then, $K(x\,|\,z) - K(y\,|\,z) = K(x\,|\,y,z) - K(y\,|\,x,z) \geq 0$, which proves the lemma. □

Theorem 6.7. *Given three strings x, y and z, the conditional normalised information distance $NID_K(x,y\,|\,z)$ satisfies the metric axioms (Definition 2.42) up to a logarithmic term.*

Proof. we need to show that $NID_K(x,y\,|\,z)$ satisfies the following distance metric axioms:

(Identity). We have: $K(x\,|\,x,z) = O(1)$ and $K(x\,|\,z) \geq 0$, implying $NID_K(x,x\,|\,z) = O(\frac{1}{K(x\,|\,z)})$.

(Symmetry). This is obvious by definition of NID_K: $NID_K(x,y\,|\,z) = NID_K(y,x\,|\,z)$.

(Triangle inequality). By Lemma 6.5,

$$\begin{aligned}\max\{K(x\,|\,y,z), K(y\,|\,x,z)\} \leq & \max\{K(x\,|\,u,z) + K(u\,|\,y,z), K(y\,|\,u,z) + K(u\,|\,x,z)\} \\ \leq & \max\{K(x\,|\,u,z), K(u\,|\,x,z)\} + \max\{K(y\,|\,u,z), K(u\,|\,y,z)\}\end{aligned}$$

and so

$$NID_K(x,y\,|\,z) \leq \frac{\max\{K(x\,|\,u,z), K(u\,|\,x,z)\}}{\max\{K(x\,|\,z), K(y\,|\,z)\}} + \frac{\max\{K(y\,|\,u,z), K(u\,|\,y,z)\}}{\max\{K(x\,|\,z), K(y\,|\,z)\}}$$

We proceed by considering the following two cases for the denominator:

Case 1: $K(u\,|\,z) \leq \max\{K(x\,|\,z), K(y\,|\,z)\}$. This implies:

$$\max\{K(x\,|\,z), K(y\,|\,z)\} \geq \max\{K(x\,|\,z), K(u\,|\,z)\} \text{ and}$$
$$\max\{K(x\,|\,z), K(y\,|\,z)\} \geq \max\{K(y\,|\,z), K(u\,|\,z)\}$$

then,

$$\begin{aligned}NID_K(x,y\,|\,z) &\leq \frac{\max\{K(x\,|\,u,z), K(u\,|\,x,z)\}}{\max\{K(x\,|\,z), K(u\,|\,z)\}} + \frac{\max\{K(y\,|\,u,z), K(u\,|\,y,z)\}}{\max\{K(y\,|\,z), K(u\,|\,z)\}} \\ &\leq NID_K(x,u\,|\,z) + NID_K(u,y\,|\,z).\end{aligned}$$

Case 2: $K(u\,|\,z) \geq \max\{K(x\,|\,z), K(y\,|\,z)\}$.

Assume without loss of generality that $K(x\,|\,z) \geq K(y\,|\,z)$. Using Lemma 6.6, we can solve the max terms and reduce the proof to:

$$\frac{K(x\,|\,y,z)}{K(x\,|\,z)} \leq \frac{K(u\,|\,x,z)}{K(u\,|\,z)} + \frac{K(u\,|\,y,z)}{K(u\,|\,z)} \tag{6.1}$$

Dividing the triangle inequality of Lemma 6.5 by $K(x\,|\,z)$, we have:

$$\frac{K(x\,|\,y,z)}{K(x\,|\,z)} \leq \frac{K(x\,|\,u,z) + K(u\,|\,y,z)}{K(x\,|\,z)}$$

Let $K(u\,|\,z) = K(x\,|\,z) + \delta$, and by symmetry of information (see Theorem 2.36): $K(u\,|\,z) + K(x\,|\,u,z) = K(x\,|\,z) + K(u\,|\,x,z)$, thus $K(u\,|\,x,z) = K(x\,|\,u,z) + \delta$.

Again we have two sub-cases:

If $\frac{K(x\,|\,u,z)+K(u\,|\,y,z)}{K(x\,|\,z)} \leq 1$, then

$$\begin{aligned}\frac{K(x\,|\,y,z)}{K(x\,|\,z)} &\leq \frac{K(x\,|\,u,z) + K(u\,|\,y,z)}{K(x\,|\,z)} \\ &\leq \frac{K(x\,|\,u,z) + K(u\,|\,y,z) + \delta}{K(x\,|\,z) + \delta} \\ &= \frac{K(u\,|\,x,z) + K(u\,|\,y,z)}{K(u\,|\,z)}.\end{aligned}$$

If on the other hand $\frac{K(x\,|\,u,z)+K(u\,|\,y,z)}{K(x\,|\,z)} \geq 1$, then we observe that $\frac{K(x\,|\,y,z)}{K(x\,|\,z)} \leq 1$ and whatever the value of $\delta \geq 0$, we will always have

$$\frac{K(u\,|\,x,z) + K(u\,|\,y,z)}{K(u\,|\,z)} = \frac{K(x\,|\,u,z) + K(u\,|\,y,z) + \delta}{K(x\,|\,z) + \delta} \geq 1.$$

In either case, inequality 6.1 is proven, and so is the theorem.

□

NID_K is mathematically exact and dimensionless [Arb11], but cannot be calculated effectively, since the Kolmogorov complexity in general is non-computable [CT06], we can only hope to calculate an approximation. As was shown in [LCL+04], it is possible to approximate Kolmogorov complexity using a lossless compressor, and subsequently, approximate information distance using

lossless compression. First, we simplify the information distance and compute its conditional complexity, using the following definitions.

Definition 6.8 (Information Distance[LCL$^+$04])**.** *The information distance $D_K(x,y)$ based on Kolmogorov complexity can be redefined as:*

$$D_K(x,y) = K(x,y) - min\{K(x), K(y)\}$$

up to an additive logarithmic term $O(\log K(x,y))$.

Similarly, we can redefine the conditional information distance.

Definition 6.9 (Conditional Information Distance)**.** *The information distance $D_K(x,y)$ conditioned to a string z, based on Kolmogorov complexity, can be redefined as:*

$$D_K(x,y\,|\,z) = K(x,y\,|\,z) - min\{K(x\,|\,z), K(y\,|\,z)\}$$

up to an additive logarithmic term $O(\log K(x,y\,|\,z))$.

The above definition can be further simplified as per the following proposition.

Proposition 6.10. *For all binary strings x, y and z.*

$$D_K(x,y\,|\,z) = K(x,y,z) - min\{K(xz), K(yz)\}$$

Proof. Using Theorem 2.36 we can write $K(x,y\,|\,z) = K(x,y,z) - K(z)$, $K(x\,|\,z) = K(xz) - K(z)$ and $K(y\,|\,z) = K(yz) - K(z)$. The conditioned information distance becomes:

$$\begin{aligned} D_K(x,y\,|\,z) &= K(x,y,z) - K(z) - \min\{K(xz) - K(z), K(xz) - K(z)\} \\ &= K(x,y,z) - K(z) - \min\{K(xz), K(xz)\} + K(z) \\ &= K(x,y,z) - \min\{K(xz), K(yz)\} \end{aligned}$$

□

Based on the above and Kolmogorov complexity approximation using real compression mechanisms, NID_K can be expressed as a normalised compression distance (*NCD*).

Definition 6.11 (Normalised Compression Distance [LCL+04])**.** *The normalised compression distance (NCD) of two binary strings x and y is defined by*

$$NCD(x,y) = \frac{C(xy) - min(C(x), C(y))}{max(C(x), C(y))}.$$

$NCD(x,y)$ is a nonnegative number belonging to $\{r \in \mathbb{R} \mid 0 \leq r \leq 1\}$ that represents how different the two binary strings are. Smaller numbers represent more similar binary strings; if $x = y$ and the compressor (C) is normal, then:

$$\begin{aligned} NCD(x,x) &= \frac{C(xx) - C(x)}{C(x)} \\ &= O\Big(\frac{\log|x|)}{C(x)}\Big) && \text{by the idempotency property of Definition 2.39} \\ &\longrightarrow 0 && \text{as } |x| \to \infty \end{aligned}$$

Therefore, if the binary sequence is large enough and the idempotency property holds up to a logarithmic term, then the identity property is preserved [CAO05]. Moreover, Cilibrasi et al. [CV05] shows that using a normal compressor (see Definition 2.39) to approximate the normalised information distance, *NCD* is a valid distance measure as it satisfies the metrics axioms.

Similarly, the conditional version of NID_K can be approximated using real compression mechanisms, as a conditional normalised compression distance.

Definition 6.12 (Conditional Normalised Compression Distance)**.** *The normalised compression distance (NCD) of two binary strings x and y conditioned to z is defined by*

$$NCD(x,y \mid z) = \frac{C(xyz) - min(C(xz), C(yz))}{max(C(xz), C(yz)) - C(z)}.$$

The conditional normalised compression trivially satisfies the identity and symmetric axioms, in a similar way to Theorem 6.7; however, it is not clear whether it satisfies the triangle inequality axiom, which will be reserved for future work.

6.3 Theoretical Metric Validation

Obfuscated programs are software in the first place. Measuring obfuscation means that, we are quantifying some software properties that may reflect the code's security. Although the security property is captured 'partially'[1] using unintelligibility according to Definition 4.1, 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 [Wey88] and Briand et al. [BMB96]. Among the proposed models, Weyuker's validation properties, despite the criticisms that were received [TZ92], have been broadly applied to certify many complexity measures, and are still an important basis and general approach to certify a complexity measure [BMB96].

Weyuker proposed nine properties or axioms for complexity validation; we will apply these properties to validate Kolmogorov complexity.[2] There are some concepts in Weyuker's properties that require some clarification in the context of Kolmogorov complexity such as functional equivalence, composition of any two programs, permutation of statements order, and renaming.

- Functional equivalence: Two programs P and Q, which belong to a set of binary strings $\{0,1\}^+$, 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. $[\![P]\!] = [\![Q]\!]$.

- 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 two programs P and Q that belong to a set of binary strings, before applying the complexity measure. $K(P,Q) = K(\langle P,Q\rangle) = K(P \parallel Q)$ up to a logarithmic term,[3] where $\parallel$ is the concatenation between programs.

- 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

[1] There are other related definition for code obfuscation based on algorithmic mutual information (see Definition 4.6).

[2] Kolmogorov complexity is approximated using compression. The validation can be, also, conducted using lossless compression instead, which yields approximately the same result.

[3] The logarithmic term is required in order to account for the computational cost finding the beginning and the end of each program.

permutation involves changes in the order or the structure of how these binary sub-strings are represented in P.[4]

- Renaming: Renaming refers to syntactic modification of a program's identifiers, variables and modules names.

Weyuker's validation properties are presented in the following.

Definition 6.13 (Weyuker's Validation Properties [Wey88])**.** *A complexity measure $C_o : P \to \mathbb{R}$ is a mapping from a program body 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 ensure the complexity measure* is not constant.

2. ***Non-coarse:*** *Given a non-negative 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 ensures that there are multiple programs of the same size.*

4. ***Functionality:*** $\exists P, Q.\ [\![P]\!] = [\![Q]\!] \wedge C_o(P) \neq C_o(Q)$. *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 increases its complexity.*

6. ***Interaction matters (a):*** $\exists P, Q, R.\ C_o(P) = C_o(Q) \wedge C_o(P \| R) \neq C_o(Q \| R)$. *This property explains the interaction of two programs of equal complexity with an auxiliary concatenated program. It states that a program R may produce different complexity measure when composed with two programs P and Q of equal complexity.*

   Interaction matters (b): $\exists P, Q, R.\ C_o(P) = C_o(Q) \wedge C_o(R \| P) \neq C_o(R \| Q)$. *This property is similar to the previous except that the identical code (R) is added at the beginning of the programs P and Q.*

7. ***Permutation is significant:*** *Let $\pi(P)$ be a permutation of P's statements. 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.*

[4]The permutations in the order of these binary sub-strings, may or may not change the semantics 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 suggested 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 by which 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 random ones, and thus a program with renamed variables might well have a different complexity than the original one. We consider this property no further.

Weyuker argued that property 9 helps to account for a situation in which a program's complexity increases as more additional components are introduced, due to the potential interaction among these parts. Briand et al. [BMB96] provided a modified version of this property (a 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 $P = m_1 \parallel m_2$ 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 6.14 (Not constant)**.** $\exists P, Q.K(P) \neq K(Q)$.

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

Proposition 6.15 (Non-coarse)**.** *Given a non-negative number c, there are only a finite number of programs such that $K(P) = c$, i.e.*

$$\exists d \in \mathbb{N}.|\{P \in \{0,1\}^+ | K(P) = c\}| \leq d.$$

Proof. According to Theorem 4.2.4 in [CT06] the number of strings, and so programs as they are binary strings too, of Kolmogorov complexity that are less than or equal to c, is upper bounded by 2^c, i.e. $|S = \{P \in \{0,1\}^+ \mid K(P) \leq c\}| \leq 2^c$, which means the set S is finite. □

Proposition 6.16 (Non-uniqueness)**.** $\exists P, Q.P \neq Q \wedge K(P) = K(Q)$.

Proof. Construct a set Q of prefix-free code with 2^n strings of length up to n, namely, Q composed only of all the strings of length n. By basic properties of Kolmogorov complexity (Theorem 2.27) the elements of Q have complexity at most $n + O(1)$ such that $\forall q \in Q.K(p) \leq n$. Therefore, by the pigeon-hole principle there must be strings with the same complexity. □

Proposition 6.17 (Functionality)**.** $\exists P, Q.[\![P]\!] = [\![Q]\!] \wedge K(P) \neq K(Q)$.

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 a program P and let $Q = P \parallel R$, 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, $[\![P]\!] = [\![Q]\!]$ and $K(Q) = K(P) + K(Q \mid P) \geq K(P)$. And because Q has to be independent from P in order to use other resources, it must be that $K(Q \mid P) = K(Q)$ and the inequality is strict. □

Proposition 6.18 (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 2.36, $K(P, Q) = K(P) + K(Q \parallel P)$. Up to a logarithmic term (see [LV08] page 663), $K(P \mid Q) = K(P, Q) = K(P) + K(Q \mid P)$. By definition, $K(Q \mid P) \geq 0$ and so $K(P) \leq K(P \parallel Q)$. The proof is similar for $K(Q)$. □

Proposition 6.19 (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 $I_K(P; Q) = O(1)$. Let $R = P$ then, we have that up to a small logarithmic factor (see [LV08] page 663) and by Theorem 2.36: $K(P \parallel R) = K(P, R) = K(P) + K(R \mid P) = K(P) + O(1)$. 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 (see Definition 2.35). If R must be different from both P and Q, then we can repeat the same proof by picking a program R that has high $I_K(P;R)$ but small $I_K(Q;R)$, for example. □

Proposition 6.20 (Permutation)**.** *Given a permutation π, $\exists P.\ K(P) \neq K(\pi(P))$.*

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 π, $Q = \pi(P)$, such that $K(Q) > K(P)$. By construction, $|P| \leq mn$ and so there are at most 2^{mn} strings with complexity smaller or equal to P. By Stirling's formula [5], $\ln n! = n \cdot \ln n - n + O(\ln n)$. Pick n such that $\ln n > \ln(2) \cdot m + 1$, which implies $n \cdot \ln n - n > \ln(2) \cdot mn$ so $n! > 2^{mn}$. Then there are more permuted programs more complex than P, and at least one permutation leads to a program more complex than P. □

Proposition 6.21 (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 2.36. 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 6.13, with two weak exceptions that have been addressed above. Therefore, we conclude Kolmogorov complexity is a suitable complexity measure for software, based on Weyuker validation framework. The below table provides a comparison, in terms of Weyuker's validation properties, between Kolmogorov complexity and the classical complexity metrics.

6.4 Measuring the Quality of Code Obfuscation

Measuring the quality of code obfuscation requires the presence of metrics that can quantify the complexity of the code. Our model evaluates the robustness of obfuscation using quantitative security metrics based on *Kolmogorov complexity* and *information distance*, which calculates how

[5] Stirling's formula is a powerful approximation for factorials.

Property	H.E.	LOC	V(G)	K
1. **Not constant**	√	√	√	√
2. **Non-coarse**	√	√	×	√
3. **Non-uniqueness**	√	√	√	√
4. **Functionality**	√	√	√	√
5. **Monotonicity**	×	√	√	√
6. **Interaction**	√	×	×	√
7. **Permutation**	×	×	×	√
8. **Renaming**	√	√	√	--
9. **Disjoint Module Additivity**	√	×	×	√

Table 6.1: Kolmogorov complexity against classical complexity measures: H.E (Halstead Effort), LOC (Lines of Code) and V(G) (Cyclomatic complexity) using Weyuker's properties

many changes to the obfuscated program a set of obfuscation transformations make, and the degree of incomprehensibility or confusion.

6.4.1 Unintelligibility Index for Code Obfuscation Incomprehensibility

In Definition 4.1 we proposed an intuitive definition for code obfuscation based on the notion of unintelligibility. Unintelligibility can be used as a metric, which measures the degree of obscurity that is introduced by a given obfuscation transformation comparing to the original code. It aims to estimate how much more difficult to understand the obfuscated code in comparison to the original code, and to which extent code obfuscation transforms the complexity of the original code. In the next definition we provide a metric for code obfuscation obscurity based on $c-$unintelligibility property of Definition 4.1 called *algorithmic unintelligibility Index*.

Definition 6.22. *The algorithmic unintelligibility Index π_U of an obfuscated code $\mathcal{O}(P,\lambda)$ is given by:*

$$\pi_U(\mathcal{O}(P,\lambda),P) = \frac{K(\mathcal{O}(P,\lambda))}{K(P)} - 1$$

It turns out that π_U is very similar to the potency metric for code obfuscation by Collberg et al. [CTL97], which used classical complexity measures to estimated code obfuscation obscurity, whereas in our case we formalise it in the context of Kolmogorov complexity using lossless compression. The approximation version of π_U is calculated using a lossless compressor C such as

$$\pi_U(\mathcal{O}(P,\lambda),P) = \frac{C(\mathcal{O}(P,\lambda))}{C(P)} - 1$$

The algorithmic unintelligibility index of unintelligible obfuscated code is always positive as the

following proposition illustrates.

Proposition 6.23. *For any $c-$unintelligible obfuscated program $\mathcal{O}(P,\lambda)$:*

$$\pi_U(\mathcal{O}(P,\lambda),P) > 0.$$

Proof. By the $c-$unintelligible of Definition 4.1, $K(\mathcal{O}(P,\lambda)) \geq (c+1)K(P)$, it is straightforward to see that $\pi_U(\mathcal{O}(P,\lambda),P) \geq 0$ for $K(\mathcal{O}(P,\lambda)) > K(P)$. □

Although π_U is always positive for unintelligible obfuscated programs, an upper bound sets a limit on π_U, which depends on the complexity of the security parameter that is used in the obfuscation process.

Proposition 6.24. *For any $c-$unintelligible obfuscated program $\mathcal{O}(P,\lambda)$:*

$$\pi_U(\mathcal{O}(P,\lambda),P) \leq \frac{K(\lambda\,|\,P)}{K(P)} + O(1)$$

Proof. By Definition 6.22

$$\pi_U(\mathcal{O}(P,\lambda),P) = \frac{K(\mathcal{O}(P,\lambda))}{K(P)} - 1$$

Using the non-information increase rule of Theorem 2.28

$$\leq \frac{K(P) + K(\lambda\,|\,P) + O(1)}{K(P)} - 1$$

$$\leq \frac{K(\lambda\,|\,P) + O(1)}{K(P)}$$

$$\leq \frac{K(\lambda\,|\,P)}{K(P)} + O\Big(\frac{1}{K(P)}\Big)$$

□

6.4.2 Normalised Kolmogorov Complexity

Kolmogorov Complexity, approximated by compression, is an absolute measure, which leads to a difficulty 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[6], and another program Q of 10^6 bits size, which is compressed to 1000 bits. By using the absolute measure of Kolmogorov

[6]Kolmogorov complexity can be seen as the length of the best compression for a given object.

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 normalised version of Kolmogorov Complexity that is relativised by the upper bound of Kolmogorov complexity i.e. the maximum complexity a certain program can achieve. Kolmogorov complexity is upper bounded by the length of a program, the subject of measure, according to Theorem 2.27; this bound can be used as the maximum Kolmogorov complexity.

Furthermore, in most cases of code obfuscation, the source code of the original program is absent, which makes the process of evaluation by comparing an obfuscated code against the original code unfeasible. Therefore, normalised Kolmogorov complexity can be useful to demonstrate the divergence of obfuscated code complexity, in terms of information content (high variability of string content), from the maximum value of that complexity, without considering the clear original code as a reference point (see Section 6.4.1 and Section 6.4.3).

Definition 6.25. *The normalised Kolmogorov complexity NK of an obfuscated code $\mathcal{O}(P,\lambda)$ is defined as:*

$$NK(\mathcal{O}(P,\lambda)) = \frac{K(\mathcal{O}(P,\lambda))}{|\mathcal{O}(P,\lambda)| + 2\log(|\mathcal{O}(P,\lambda)|)}$$

Where $|\mathcal{O}(P,\lambda)|$ is the length of $\mathcal{O}(P,\lambda)$.

NK is normalised to take a value between 0 and 1 by dividing the Kolmogorov complexity of obfuscated code with the upper bound (maximum value that can be achieved by Kolmogorov complexity) in Theorem 2.27. 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 obfuscated code. As before,*NK* is estimated using compression; an approximate version of *NK* is denoted by *NC*:

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

NC is non-negative number that belongs to $\{r \in \mathbb{R} \mid 0 \leq r \leq 1\}$. We assume that if the size of compressed obfuscated code is equal to the size of obfuscated code itself, then the obfuscated code is highly random, and is difficult to comprehend by an attacker. This can be justified in light of our discussion of code regularity in Section 4.2.

This is particularly true, as argued in [STZDG14], for relatively large programs (binary strings of large length), since for very small programs the overhead necessary to encode the regularities of a program could make the result larger than the original. Given that regularities can be encoded by a relatively small piece of code, the larger the number and size of these regularities the greater the savings of the compressed form will be.

6.4.3 Information Distance for Code Obfuscation Resilience

The unintelligibility index π_U suffers from the same problem we discussed in Example 4.4. A high unintelligibility index does not always imply secure code obfuscation, it really depends on how the security parameter is blended with the original code. In this section we introduce information distance as a potential solution for this problem, and establish a link with our definition of obfuscation security in Definition 4.6.

The information distance metric can serve as a basis to quantify the level of protection that is provided by obfuscating programs. Intuitively, the deobfuscated program should be obscure enough compared to the original program, which gives a general indication of the resilience of the obfuscation method against an adversary equipped with deobfuscation tools, and the level of confusion an obfuscated technique added.

What makes information distance an interesting approach to measure software, in general, and code obfuscation, in particular, is that it has a solid mathematical foundation. In mathematics a metric is called a *measure of distance* if it satisfies the metric axioms (see Definition 2.42). Despite the fact that the information distance is a metric, it does not provide a straightforward intuition for measuring the security in obfuscated programs. There are many issues that need to be resolved in order to consider information distance a valid measure for code obfuscation i.e. we need to establish a logical and theoretical link between our definitions for code obfuscation security and information distance, and need to specify under which conditions we can use it.

First, we observe that we can use a simplified definition for NID_K when the obfuscated code adds even a modest amount of complexity.

Lemma 6.26. *For an obfuscated code $\mathcal{O}(P, \lambda)$, satisfying unintelligibility property of Definition 4.2,*

$K(\mathcal{O}(P,\lambda)\,|\,P) > K(P\,|\,\mathcal{O}(P,\lambda))$ *up to an additive constant, and*

$$NID_K(P,\mathcal{O}(P,\lambda)) = \frac{K(\mathcal{O}(P,\lambda)|P)}{K(\mathcal{O}(P,\lambda))}$$

Proof. By the chain rule of Theorem 2.36 up to a logarithmic term,

$$K(P) - K(P\,|\,\mathcal{O}(P,\lambda)) = K(\mathcal{O}(P,\lambda)) - K(\mathcal{O}(P,\lambda)\,|\,P)$$

By c-unintelligibility, $K(\mathcal{O}(P,\lambda)) > K(P)$, so

$$K(\mathcal{O}(P,\lambda)) - K(P\,|\,\mathcal{O}(P,\lambda)) > K(\mathcal{O}(P,\lambda)) - K(\mathcal{O}(P,\lambda)\,|\,P)$$

Therefore,

$$K(\mathcal{O}(P,\lambda)\,|\,P) > K(P\,|\,\mathcal{O}(P,\lambda)) \tag{6.2}$$

It is straightforward to compute the NID_K

$$\begin{aligned} NID_K(P,\mathcal{O}(P,\lambda)) &= \frac{\max\{K(\mathcal{O}(P,\lambda)\,|\,P), K(P\,|\,\mathcal{O}(P,\lambda))\}}{K(\mathcal{O}(P,\lambda))} && \text{by Definition 6.3} \\ &= \frac{K(\mathcal{O}(P,\lambda)\,|\,P)}{K(\mathcal{O}(P,\lambda))} && \text{by Eq. (6.2)} \end{aligned}$$

□

We proceed by illustrating the relation between secure obfuscated code and normalised information distance as a quantitative metric for code obfuscation resilience. We give an upper bound to this metric, and show that it is closely met by a secure code obfuscation, by providing the following theorem.

Theorem 6.27. *Let $\mathcal{O}(P,\lambda)$ be a c−unintelligible obfuscated program for $c > 0$. Then, the normalised information distance between P and $\mathcal{O}(P,\lambda)$ is upper bounded by $1 - \frac{I_K(P;\mathcal{O}(P,\lambda))}{K(\mathcal{O}(P,\lambda))}$. Furthermore, if $\mathcal{O}(P,\lambda)$ is ϵ-secure according to Definition 4.6, then the normalised information distance approaches 1 as ϵ approaches 0.*

Proof. By Lemma 6.26 and Definition 4.1

$$NID_K(P, \mathcal{O}(P,\lambda)) = \frac{K(\mathcal{O}(P,\lambda)\,|\,P)}{K(\mathcal{O}(P,\lambda))}$$

$$= \frac{K(\mathcal{O}(P,\lambda)) - K(\mathcal{O}(P,\lambda)) + K(\mathcal{O}(P,\lambda)\,|\,P)}{K(\mathcal{O}(P,\lambda))}$$

By Definition 2.35

$$= 1 - \frac{I_K(P;\mathcal{O}(P,\lambda))}{K(\mathcal{O}(P,\lambda))}$$

which proves the upper bound. Now consider the assumptions: $I_K(P;\mathcal{O}(P,\lambda)) \leq \epsilon K(P)$ and $K(\mathcal{O}(P,\lambda)) \geq (c+1)K(P)$. These lead to

$$\begin{aligned}
\frac{I_K(P;\mathcal{O}(P,\lambda))}{K(\mathcal{O}(P,\lambda))} &\leq \frac{\epsilon K(P)}{K(\mathcal{O}(P,\lambda))} \\
&\leq \frac{\epsilon K(\mathcal{O}(P,\lambda))}{(c+1)K(\mathcal{O}(P,\lambda))} \\
&= \frac{\epsilon}{c+1} \\
&\longrightarrow 0 \qquad\qquad \text{As } \epsilon \longrightarrow 0.
\end{aligned}$$

Given that ϵ is a small number lesser than 1, NID_K is approximately equal to one, which is the maximum distance. □

So far, we established the relation between our security definition and the normalised information distance. We showed that if the security conditions for code obfuscations are satisfied (the extreme case) then the normalised information distance tends to be close to one, which is the maximum value that can be achieved, which also shows that the obfuscated code is totally different than the original unobfuscated version.

In the following we are presenting some bounds on the value of normalised information distance for obfuscated code. We can derive an upper bound based on the complexity of the obfuscation parameter and its relation to the obfuscated code, as we see in the next theorem.

Theorem 6.28. *For an obfuscated c-unintelligible program $\mathcal{O}(P,\lambda)$, where $c > 0$, the normalised information distance is upper bounded:*

$$NID_K(P, \mathcal{O}(P,\lambda)) \leq \frac{K(\lambda)}{K(\mathcal{O}(P,\lambda))} + O\Big(\frac{1}{K(\mathcal{O}(P,\lambda))}\Big)$$

Proof.

$$
\begin{aligned}
NID_K(P, \mathcal{O}(P,\lambda)) &= \frac{K(\mathcal{O}(P,\lambda)\,|\,P)}{K(\mathcal{O}(P,\lambda))} && \text{by Lemma 6.26}\\
&\leq \frac{K(\lambda\,|\,P) + O(1)}{K(\mathcal{O}(P,\lambda))} && \text{by Lemma 4.11}\\
&\leq \frac{K(\lambda) + O(1)}{K(\mathcal{O}(P,\lambda))} && \text{by Theorem 2.27.}
\end{aligned}
$$

□

We notice from the above theorem that normalised information distance for an obfuscated program depends on the amount of information is the security parameter (complexity). If the security parameter is available to an adversary $\mathcal{A}$, then clearly the normalised information distance can be the minimum as the next theorem shows.

Theorem 6.29. *Let $\mathcal{O}(P,\lambda)$ be an obfuscated program, for an adversary $\mathcal{A}$ satisfying the intelligibility property of Definition 5.1, the normalised information distance conditioned on the security parameter λ is :*

$$NID_K(P, \mathcal{O}(P,\lambda))\,|\,\lambda) = O\Big(\frac{\log n}{K(P\,|\,\lambda)}\Big)$$

where n is the length of P and λ.

Proof.

$$
\begin{aligned}
NID_K(P, \mathcal{O}(P,\lambda)|\lambda) &= \frac{\max\{K(P\,|\,\mathcal{O}(P,\lambda),\lambda), K(\mathcal{O}(P,\lambda)\,|\,P,\lambda)\}}{\max\{K(P\,|\,\lambda), K(\mathcal{O}(P,\lambda)\,|\,\lambda)\}} && \text{by Definition 6.4}\\
&= \frac{K(P,\mathcal{O}(P,\lambda)\,|\,\lambda) - \min\{K(P\,|\,\lambda), K(\mathcal{O}(P,\lambda)\,|\,\lambda)\}}{\max\{K(P\,|\,\lambda), K(\mathcal{O}(P,\lambda)\,|\,\lambda)\}} && \text{by Definition 6.9}\\
&= \frac{K(P,\mathcal{O}(P,\lambda)\,|\,\lambda) - K(\mathcal{O}(P,\lambda)\,|\,\lambda)}{K(P\,|\,\lambda)} && \text{by Theorem 2.29}\\
&= \frac{K(P\,|\,\lambda) - K(P\,|\,\lambda) + K(P,\mathcal{O}(P,\lambda)\,|\,\lambda) - K(\mathcal{O}(P,\lambda)\,|\,\lambda)}{K(P\,|\,\lambda)}\\
&= \frac{K(P\,|\,\lambda) - I_K(P,\mathcal{O}(P,\lambda)\,|\,\lambda)}{K(P\,|\,\lambda)} && \text{by Definition 2.35}\\
&= \frac{K(P\,|\,\lambda) - K(P\,|\,\lambda) + O(\log n)}{K(P\,|\,\lambda)} && \text{by Lemma 5.4}\\
&= O\Big(\frac{\log n}{K(P\,|\,\lambda)}\Big)
\end{aligned}
$$

□

So far we established an upper bound on the normalised information distance; in the next theorem we proceed with investigating the existence of a lower bound, which sets a threshold on obfuscated code security using NID_K.

Theorem 6.30. *For an obfuscated program $\mathcal{O}(P,\lambda)$, assume λ is optimal such that: $K(\mathcal{O}(P,\lambda)\,|\,P) \geq K(\lambda\,|\,P) - \alpha$, for some $\alpha \in \mathbb{N}$. Then the normalised information distance has a lower bound:*

$$NID_K(P,\mathcal{O}(P,\lambda)) \geq \frac{K(\lambda\,|\,P) - \alpha}{K(\mathcal{O}(P,\lambda))}$$

Proof. By Lemma 6.26

$$NID_K(P,\mathcal{O}(P,\lambda)) = \frac{K(\mathcal{O}(P,\lambda)\,|\,P)}{K(\mathcal{O}(P,\lambda))}$$

It follows from the assumption of the optimality of λ that

$$NID_K(P,\mathcal{O}(P,\lambda)) \geq \frac{K(\lambda|P) - \alpha}{K(\mathcal{O}(P,\lambda))}$$

□

The intuition behind the optimality assumption was justified in Lemma 4.11, where we showed that λ must have the minimum algorithmic information content to go from $\mathcal{O}(P,\lambda)$ to P.

6.4.4 Normalised Information Distance for Individual Security

In Section 4.3 we identified two main types of code obfuscation (Encoding and Hiding) and studied their security based on individual security level i.e. the security of obfuscated sub-programs. We further extend these results by studying the effect of NID_K when the obfuscator employs encoding and hiding techniques to obfuscate.

It is interesting to ask the following question: can we reason about the security of the obfuscated program as a whole set of subprograms, and its NID_K? In order to check the total security of each individual obfuscated subprogram in the set Q, we turn to a notion borrowed from classical information theory called *Channel Capacity* [CT06], which computes the maximum capacity a channel can have over all source distributions. Since the channel is the mutual information in Classical Information Theory and we use algorithmic version of mutual information in our theory,

we find it is natural and intuitive to require a maximization over all the individual sub-programs of original code for algorithmic mutual information. That helps to capture the maximum possible leakage (insecurity) of clear subprogams from the obfuscated ones. The following definitions capture these intuitions.

Definition 6.31 (Total Hiding)**.** *Let Q be a collection of obfuscated subprograms $\mathcal{O}(p_i, \kappa_i)$, where $p_i \in P$ and $\kappa_i \in \lambda$, we define the total hiding security of Q as the maximum algorithmic mutual information among all clear code such that:*

$$InSec_H(Q) = max_{p_i \in P, \mathcal{O}(p_i,\kappa_i) \in Q}\{I_K(p_i; \mathcal{O}(p_i, \kappa_i))\}$$

Similarly, we can define the total security in case of obfuscation encoding.

Definition 6.32 (Total Encoding)**.** *Let Q be a collection of obfuscated subprograms $\mathcal{O}(p_i, \kappa_i)$, where $p_i \in P$ and $\kappa_i \in \lambda$, we define the total encoding security of Q such that:*

$$InSec_E(Q) = max_{\kappa_i \in \lambda, \mathcal{O}(p_i,\kappa_i) \in Q}\{I_K(\kappa_i; \mathcal{O}(p_i, \kappa_i))\}$$

In the above definitions we are looking at the maximum leakage (insecurity) in obfuscated code with respect to the original code i.e. we are checking the worst-case scenario to reason about all obfuscated subprograms in a obfuscated set that resembles the whole obfuscated program. The obfuscated sub-program that leaks the most about the original one, among all other obfuscated subprograms, is the weakest link in the security chain. The following lemma is a direct result of applying Theorem 4.15 and Theorem 4.14 on the individual security level.

Lemma 6.33. *Let Q be a collection of obfuscated subprograms $\mathcal{O}(p_i, \kappa_i)$, where $p_i \in P$ and $\kappa_i \in \lambda$, then:*

$$InSec_H(Q) \leq max_{p_i \in Q}\{\delta_{P_i}\} - O(1) \text{ and}$$
$$InSec_E(Q) \leq max_{\kappa_i \in Q}\{\delta_{\kappa_i}\} - O(1)$$

Proof. It can be proved directly by :

$$\begin{aligned}
InSec_H(Q) &\leq \max_{p_i \in P, \mathcal{O}(p_i,\kappa_i)\in Q}\{I_K(p_i; \mathcal{O}(p_i,\kappa_i))\} && \text{by Definition 6.31} \\
&\text{since } p_i \text{ is hidden in } Q \text{ but in its original form, and by Theorem 4.15} \\
&\leq \max_{p_i \in Q}\{\delta_{P_i} - O(1)\} \\
&\leq \max_{p_i \in Q}\{\delta_{P_i}\} - O(1)
\end{aligned}$$

Similarly,

$$\begin{aligned}
InSec_E(Q) &\leq \max_{\kappa_i \in \lambda, \mathcal{O}(p_i,\kappa_i)\in Q}\{I_K(\kappa_i; \mathcal{O}(p_i,\kappa_i))\} && \text{by Definition 6.32} \\
&\text{the subkey } \kappa_i \text{ is hidden in } Q \text{ } (\kappa_i \text{ contains the encoding rules), and by Theorem 4.14} \\
&\leq \max_{\kappa_i \in Q}\{\delta_{\kappa_i} - O(1)\} \\
&\leq \max_{\kappa_i \in Q}\{\delta_{\kappa_i}\} - O(1)
\end{aligned}$$

□

In Theorem 6.27 we established a relation between algorithmic mutual information for code obfuscation security and the normalised information distance. Intuitively, this relation states that a reduction in the algorithmic mutual information produces an increase in the normalised information distance between a program and its obfuscated version. Now, the question that can be raised here is: whether can we establish the same relation, as in Theorem 6.27, based on the security of individual level, i.e. sub-programs? In order to answer this question, we need first to compute the overall information distance of all distances of clear sub-programs and their obfuscated versions on the individual security level; secondly, we have to check for the existence of such a relation.

The overall normalised information distance, in case of individual sub-programs, is computed by finding the minimum value among all distances between clear and obfuscated sub-programs. Using the same reasoning as in Definition 6.31 and Definition 6.32 of the worst case scenario for overall obfuscation security, we require the overall distance to be minimum. The smallest distance among all clear sub-programs and their obfuscated versions indicates the weakest chain in the security of the whole obfuscated sub-programs.

Definition 6.34 (Total Hiding Distance)**.** *The overall distance $HNID_K$ between a set of subpro-*

grams $p_i \in P$ and their obfuscated versions $\mathcal{O}(p_i, \kappa_i) \in Q$ is defined as

$$HNID_K(Q) = min_{p_i \in P, \mathcal{O}(p_i,\kappa_i) \in Q}\{NID_K(p_i, \mathcal{O}(p_i, \kappa_i))\}$$

The above definition addresses the total hiding distance in an obfuscated program consisting of a collection of sub-programs, where the original code still exists in the obfuscated version but in a hidden state (see Section 4.3). The next definition is similar, but it tackles the total encoding distance where the encoding process (security parameter) is hidden, instead of the original sub-program, in the obfuscated set of sub-programs.

Definition 6.35 (Total Encoding Distance)**.** *The overall distance $ENID_K$ between a set obfuscated subprograms $\mathcal{O}(p_i, \kappa_i) \in Q$ and their security parameters $\kappa_i \in \lambda$ is defined as*

$$ENID_K(Q) = min_{\kappa_i \in \lambda, \mathcal{O}(p_i,\kappa_i) \in Q}\{NID_K(\kappa_i, \mathcal{O}(p_i, \kappa_i))\}$$

In the next stage we investigate the relation between overall obfuscation security on the individual level and the total distance for hiding and encoding obfuscation techniques.

Theorem 6.36 (Maximum Hiding Distance)**.** *Consider Q a set of obfuscated subprograms $\mathcal{O}(p_i, \kappa_i)$ of $p_i \in P$ using security parameters $\kappa_i \in \lambda$, then:*

$$HNID_K(Q) \geq 1 - \frac{max_{p_i \in Q}\{\delta_{p_i}\}}{min_{\mathcal{O}(p_i,\kappa_i) \in Q}\{K(\mathcal{O}(p_i, \kappa_i))\}}$$

Proof. By Definition 6.35:

$$\begin{aligned} HNID_K(Q) &= \min_{p_i \in P, \mathcal{O}(p_i,\kappa_i) \in Q}\{NID_K(p_i, \mathcal{O}(p_i, \kappa_i))\} \\ &\quad \text{By Lemma 6.26, and Definition 4.2} \\ &= \min_{p_i \in P, \mathcal{O}(p_i,\kappa_i) \in Q}\left\{\frac{K(\mathcal{O}(p_i, \kappa_i) \,|\, p_i)}{K(\mathcal{O}(p_i, \kappa_i))}\right\} \\ &\quad \text{By Definition 2.35} \\ &= \min_{p_i \in P, \mathcal{O}(p_i,\kappa_i) \in Q}\left\{1 - \frac{I_K(p_i; \mathcal{O}(p_i, \kappa_i))}{K(\mathcal{O}(p_i, \kappa_i))}\right\} \end{aligned}$$

$$= 1 - \max_{p_i \in P, \mathcal{O}(p_i,\kappa_i) \in Q} \left\{ \frac{I_K(p_i; \mathcal{O}(p_i, \kappa_i))}{K(\mathcal{O}(p_i, \kappa_i))} \right\}$$

$$= 1 - \frac{\max_{p_i \in P, \mathcal{O}(p_i,\kappa_i) \in Q} \{ I_K(p_i; \mathcal{O}(p_i, \kappa_i)) \}}{\min_{p_i \in Q, \mathcal{O}(p_i,\kappa_i) \in Q} \{ K(\mathcal{O}(p_i, \kappa_i)) \}}$$

by Definition 6.32 and Lemma 6.33-hiding

$$\geq 1 - \frac{\max_{p_i \in Q} \{ \delta_{p_i} \}}{\min_{\mathcal{O}(p_i,\kappa_i) \in Q} \{ K(\mathcal{O}(p_i, \kappa_i)) \}}$$

□

Theorem 6.37 (Maximum Encoding Distance)**.** *Consider Q a set of obfuscated subprograms $\mathcal{O}(p_i, \kappa_i)$ of $p_i \in P$ using random security parameters $\kappa_i \in \lambda$, then:*

$$ENID_K(Q) \geq 1 - \frac{max_{\kappa_i \in \lambda} \{ \delta_{\kappa_i} \}}{min_{\kappa_i \in Q} \{ K(\kappa_i) \}}$$

Proof. By assumption on security parameters randomness, $K(\mathcal{O}(p_i, \kappa_i)) \leq K(\kappa_i)$[7]. Then we follow the same proof steps as with Theorem 6.36. □

6.4.5 Stealth of Code obfuscation

A typical software application can be very large, often millions of lines of code. For this reason an initial step in any attack against software is to attempt to isolate code-segments that are more likely than others to contain security-sensitive code. This classification can be based on the location of the code, the order in which it is being executed (code executed early on is more likely to contain security checks), whether it contains unusual code sequences, etc. It is therefore essential that an obfuscated code, which is inserted into an application is stealthy, so that it does not arouse any attention. For example, an algorithm that using a number of *xor* instructions to obfuscate a program would likely be obvious, since most real programs contain very few, if any, *xors*. An obfuscated code needs to be stealthy in two ways:

1. **Intrinsic Stealth:** The obfuscated code should be similar to the code that surrounds it.

2. **Extrinsic Stealth:** The obfuscated code should be similar to the code that occurs in a typical application.

[7]The obfuscated programs are not as random as the (secret) security parameters (keys) that are used to obfuscate.

The first condition makes it difficult for the attacker to select some particular methods in which there is more chance of occurrence of the obfuscated code. The later condition ensures that the obfuscated application, in its entirety, is less likely to throw suspicion of a presence of obfuscation. Both the conditions makes its difficult to carry out manual, as well as automated attacks, in order to determine the presence and the specific location(s) of the obfuscated code.

Stealth defines how well the obfuscated code fuses with the rest of the program. Stealth of obfuscated code, intuitively, is related to the attacker's ability to distinguish the obfuscated code that is located in a given entity. The stealth of interest here, is self-stealth (intrinsic) of the obfuscated code to the surrounding code where it is located i.e. how well an obfuscated code is hidden. If an obfuscation transformation introduces code that stands out from the rest of the program, it may be difficult for an automatic deobfuscator to spot it, but it can easily spotted by a reverse engineer. Obvious obfuscation offers reverse engineers a clue to identify which obfuscation technique is applied to the original code, because each technique has special characteristics. For example, inserting 'junk' bytes introduces many invalid instructions observed rarely in a normal binary, and encrypting the original code introduces many data bytes observed rarely in an executable binary. This measure is an important factor for code obfuscation security; stealthy obfuscated code can enhance the protection level so that attackers should put more effort to figure out the used obfuscation techniques and where the obfuscated parts are located in the code.

It is intuitive and natural to think about some sort of similarity measures to estimate and measure the stealth in code obfuscation. Checking whether an obfuscated code is stealthy is very similar to detecting malware. Similarity distance was used extensively to detect malware [LXX+09] [ABCD15] [CX12] including information distance and its approximation (*NCD*). Therefore, it could be possible to use NID_K and *NCD* to measure the stealth of obfuscated code. However, Zhang et al demonstrate in [ZHZ+07], that the conventional information distance, and its normalised version (NID_K) has a problem. It can account for irrelevant information that overwhelm the similarity result. This irrelevant information is an issue when perform 'partial pattern' matching between two objects. Measuring the stealth of obfuscating code is based on finding any matched code in the surrounding programs with the obfuscated code, without including all the irrelevant information; therefore, NID_K and *NCD* cannot be very helpful to detect which part of software is obfuscated. We highlight this problem by providing the following example:

Example 6.38. *Consider an obfuscated sub-program q in a set of sub-programs Q, which contains* similar *sub-programs to q, and also have some other sub-programs that are different to q. We need to check whether q is stealthy with respect to Q. We expect the similarity distance measure to answer this question. Intuitively, q is similar to some sub-programs in Q by definition, so we expect the similarity distance to report that q is stealthy with respect to Q.*

For convenience, we compute the similarity version instead, denoted by SNID, by subtracting NID_K from 1. Given two binary string $x, y \in \{0,1\}^+$:

$$SNID_k(x,y) = 1 - \frac{max\{K(x\,|\,y), K(y\,|\,x)\}}{max\{K(x), K(y)\}}$$

We compute the similarity distance to measure the degree of stealth of q in Q.

$$SNID_k(q,Q) = 1 - \frac{max\{K(q\,|\,Q), K(Q\,|\,q)\}}{max\{K(q), K(Q)\}}$$

Since $q \in Q$ then by Theorem 2.36 $K(q) \leq K(Q)$,

and by Lemma 6.26 $K(q\,|\,Q) \leq K(Q\,|\,q)$, then

$$= 1 - \frac{K(Q\,|\,q)}{K(Q)}$$

$$= \frac{K(Q) - K(Q,q) + K(q)}{K(Q)} \text{ by Theorem 2.36-1}$$

$K(q,Q) = K(Q)$ as $q \in Q$ and by Theorem 2.36-1, then we have

$$= \frac{K(q)}{K(Q)}$$

The similarity distance, according to our derivations, depends on $K(Q)$; it shows that the result is far less than 1, as $K(Q)$ is much bigger than $K(q)$ (q is a sub-program belongs to the set Q). However, this contradicts our assumption that q is an obfuscated program similar to other sub-programs in Q, which means $SNID_k$ failed to report the right value; it must report a high value close to 1 (normalised). Furthermore, if Q contains irrelevant information (other sub-programs that are different from q), then it makes $SNID_k$ accounts for these information. This could indicate that q is unstealthy with respect to Q. However, an attacker would not be able to tell if q is obfuscated or not, because by comparing q to Q s/he might find a similar chunks of code in Q that makes q looks unsuspicious (stealthy).

We conclude from the above example that the conventional information distance NID_K and its

approximated version *NCD* could be inefficient measuring the stealth of obfuscated code, this is apparently due to the potential existence of irrelevant information in the surrounding code that could overwhelm the results. In order to go around this problem, we need to only consider the information that is relevant for our stealth matching i.e. we need to minimise measuring the amount of information in the surrounding code that is normally ignored by attacker, when s/he performs partial matching attack.

In [ZHZ+07] the authors introduced and justified an extension to NID_K that solves the problem of partial pattern matching, It only requires the minimum amount of irrelevant information for comparison. The new distance is defined and normalised as follows:

Definition 6.39 (Normalised Minimum Information Distance [ZHZ+07])**.** *The normalised minimum information distance is defined as*

$$NID_{min}(x,y) = \frac{min\{K(x\,|\,y), K(y\,|\,x)\}}{min\{K(x), K(y)\}}$$

In order to minimize the irrelevant information between obfuscated code and the surrounding code, and to measure the degree of stealthiness, we will rely on NID_{min} to obtain such a result. We can also derive a similarity version of NID_{min} by subtracting from 1.

Definition 6.40. *The similarity version of normalised minimum information distance is defined as*

$$SNID_{min}(x,y) = 1 - \frac{min\{K(x\,|\,y), K(y\,|\,x)\}}{min\{K(x), K(y)\}}$$

Using $SNID_{min}$, we can solve the problem we discussed in Example 6.38, using the same derivation steps:

$$\begin{aligned} SNID_{min}(x,y) &= 1 - \frac{\min\{K(q\,|\,Q), K(Q\,|\,q)\}}{\min\{K(q), K(Q)\}} \\ &= 1 - \frac{K(q\,|\,Q)}{K(q)} \\ &= \frac{K(q) - K(Q,q) + K(Q)}{K(q)} \\ & K(q,Q) = K(Q) \text{ as } q \in Q, \text{ then} \\ &= \frac{K(q)}{K(q)} = 1. \end{aligned}$$

NID_{min} can be approximated in a similar way to *NID* using compression, such that:

$$NCD_{min} = \frac{C(xy) - \max(C(x), C(y))}{\min(C(x), C(y))}.$$

Similarly, we can derive a similarity distance version of NCD_{min}:

$$\begin{aligned} SNCD_{min} &= 1 - \frac{C(xy) - \max(C(x), C(y))}{\min(C(x), C(y))} \\ &= \frac{\min((C(x), C(y)) - C(xy) + \max(C(x), C(y))}{\min(C(x), C(y))} \\ &= \frac{C(x) + C(y) - C(xy)}{\min(C(x), C(y))} \end{aligned}$$

We notice that NID_{min} minimises the effect of irrelevant information by taking the minimum difference between the two objects. The main aim of introducing NID_{min} is to solve the problem of irrelevant information between two objects; however, this measure is not a full metric as it does not satisfy the triangle inequality axiom. Li et al in [ZHZ+07] and Fagin and Stockmeyer in [FS98] argued using two different examples that in many cases for similarity measure, especially for partial pattern matching, it is not necessarily *must* for triangle inequality to hold.

The above reasoning and discussion leads us to provide a new definition for code obfuscation stealth based on the modified version of information distance. Intuitively, it means that the stealth of a given obfuscated code (whether intrinsic or extrinsic) requires the similarity between the obfuscated code and its surrounding code to not go beyond a certain threshold.

Definition 6.41. *An obfuscated program q is δ-stealth with respect to a set of programs Q and an adversary $\mathcal{A}$, if $SNID_{min}$ is lower bounded by δ:*

$$SNID_{min}(q, Q) \geq \delta$$

Where $0 < \delta \leq 1$.

6.4.6 A Statistical Model for Code Obfusction Metrics

As has been discussed, measuring code obfuscation presents a serious challenge. We argue that no single metric can provide a powerful enough predictive model to estimate the total protection

provided by code obfuscation. We are aiming to derive a minimal set of easily calculated metrics that can produce a single value, which quantifies and measures the quality of code obfuscation. The proposed metrics are by no means complete to compute the robustness of code obfuscation, and we do not claim it is the best overall. It presents an important milestone toward building a sufficient and complete metrics to quantify code obfuscation.

A Regression model [KKM88], which is a statistical method for estimating the relationships among variables (discussed in Section 7.5 and in more details in Section 8.2.4), can be used to produce a polynomial equation (linear regression equation) which takes our metrics as independent variables that contribute to overall code obfuscation robustness, i.e. the proposed metrics are used as factors (variables) which impact the security of obfuscated software, and then a regression model is built to calculate the security level.

The security of code obfuscation can be quantified and measured using this list of proposed measures : normalised information distance, unintelligibility index, normalised Kolmogorov complexity and distance stealth measure.

Definition 6.42. *Given an obfuscated program $\mathcal{O}(P,\lambda)$, the* Total Security quantity*, $S_q^{P,\mathcal{O}(P,\lambda)}$ is defined as:*

$$S_q^{P,\mathcal{O}(P,\lambda)} = (NID_K(P,\mathcal{O}(P,\lambda)); \pi_U(P,\mathcal{O}(P,\lambda)); NK(\mathcal{O}(P,\lambda)); SNID_{min}(P,\mathcal{O}(P,\lambda)))$$

NID_K represents the information distance (based on normalised compression distance) (Definition 6.3), π_U is the unintelligibility index measure (Definition 6.22) and NK is the normalised Kolmogorov complexity (Definition 6.25), where $SNID_{min}$ (Definition 6.40) is the distance stealth measure.

Having the above metrics, we can turn the quadruple values into a linear regression equation [She07], which can predict the security of code obfuscation:

$$S_z^{P,\mathcal{O}(P,\lambda)} \triangleq a_0 + a_1 * NID_K + a_2 * \pi_U + a_3 * NK + a_4 * SNID_{min}$$

Where a_0 is the regression intercept, and a_1, a_2 and a_3 are the regression coefficient or parameters (weight of the effect factor). The intercept value can be interpreted as the value of $S_q^{P,\mathcal{O}(P,\lambda)}$ when a_1, a_2 and a_3 are zeros. The regression coefficient a_1 can be explained as for every unit changes

(increase or decrease) in NID_K, we expect a_1 change in $S_q^{P,\mathcal{O}(P,\lambda)}$, holding all other variables constant.

This equation provides a way to measure and reason about the security of different obfuscation techniques. All the independent variables in S_q are based on uncomputable version of Kolmogorov complexity which can be approximated using lossless compression. The new estimated version of S_q is denoted by $\hat{S}_q$ and is computed as follow:

$$\hat{S}_q^{P,\mathcal{O}(P,\lambda)} = \hat{a}_0 + \hat{a}_1 * NCD + \hat{a}_2 * \pi_U + \hat{a}_3 * NC + \hat{a}_4 * SNCD_{min}$$

All these parameters are estimated empirically. In the next chapter, we conduct an experiment on a set of Java bytecode programs using a wide range of code obfuscation techniques, to calculate the proposed metrics and derive the regression model coefficients.

6.5 Summary

In this chapter we proposed a theoretical model for measuring the quality of code obfuscation. We provided a theoretical evaluation for Kolmogorov complexity based on Weyuker's validation properties. We showed that Kolmogorov complexity is a suitable metric for measuring complexity in binary programs, and code obfuscation in particular. We also adapt the work of Li et al. on the notion of information distance for measuring similarity between clear code and its obfuscated version. We extend their theoretical work to reason about the quality of code obfuscation by relating information distance to the security in code obfuscation. We formalise the notion of unintelligibility index (degree of confusion introduced) and the relative Kolmogorov complexity, and showed that information distance metrics is a suitable measure for code obfuscation. We also used a modified version of information distance to define code obfuscation stealth, and propose a statistic model to measure the total security of obfuscation based on linear regression model.

7 Experimental Design and Tool-sets

In this chapter we present our experimental design and tool-sets that we used in the evaluation of our metrics. We followed Wohline et al.'s framework [WRH+00] on experimental software engineering to design and analyse the findings. The aim is to provide detailed and comprehensive information about the experiment's design and the tools which are necessary to conduct and interpret the obtained results. We applied the experiment on a set of obfuscated Java jar files of the SPECjvm2008 benchmark, using two obfuscators: Sandmark, an open source suite, and Dasho, a commercial tool. We choose three different decompilers as an attack model, to assess the level of resilience of obfuscated programs. We present the research problem as a set of research questions, each question is formulated with a null hypothesis, which will be answered by using a set of statistical methods.

7.1 Introduction

Recent empirical results [JF14] show that data compression is a promising software metric technique to estimate human comprehension in software. So far, we have presented a theoretical validation that complements this result, and we proposed a formal definition that reflects the natural intuition of code obfuscation's unintelligibility. We further advanced on the unintelligibility notion to define the security of code obfuscation based on algorithmic mutual information (see Definition 4.6) with respect to a specific adversary model (see Definition 5.1). Then, based on the proposed theoretical foundation, we derived a set of metrics to quantitatively measure the quality of code obfuscation, including: normalised compression distance, unintelligibility index and normalised compression measure.

Despite the importance theoretical validation for metrics, in general, and code obfuscation metrics, in particular, the measure is meant to reflect the empirical characteristic of the software's

properties. For example, if an obfuscated code P is more secure than other obfuscated code Q, then the measure has to reflect this security. Therefore, it is necessary to validate any proposed measure empirically. The measure is validated by showing that it correlates with some other corresponding factors or other existing measure. The corresponding factor is chosen based on the wide, intuitive acceptance or on reasonable assumptions, which makes the validation factor itself 'valid' by definition. Although it is always advantageous to use the validation factor directly, it is normally not feasible to do so or expensive to conduct it. Therefore, the proposed measure represents an easy and effective way.

To provide the empirical validation we conducted an experiment, using the proposed metrics, 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, an open source suite, and Dasho, a commercial tool. Moreover, we employed three decompilers as a model of attack to study the resilience of code obfuscation. The experimental work is spread over two chapters; in this chapter we show how we design the experiment, and what tool-sets are used. We state the research questions and what methodology to apply in order to answer these questions. The next chapter is concerned with presenting and interpreting the experimental results.

The remainder of this chapter is structured as follows. In Section 7.2, we present the scope of the experiment, which includes the experiments objectives. Section 7.3 provides the experimental design and set-up, experimental process, and the analysis methodology.

7.2 Scope of the experiment

The experimental design and empirical evaluation was adapted from Wohline et al.'s framework on experimental software engineering [WRH$^+$00]. The goal of this experiment is to provide empirical evidence that our metrics are suitable for measuring the quality of code obfuscation.

7.2.1 Objectives

In general, the experiment is concerned with measuring the obfuscation resilience against a set of deobfuscators (decompilers). The main aim of conducting this experiment is two-fold: the first goal is to study and validate the usefulness of our proposed model, as an appropriate quantitative measure

for code obfuscation quality against a certain attack. Secondly, to evaluate the effectiveness of code obfuscation using the proposed metric, i.e. analyse the effect of code obfuscation on security using the proposed metrics.

In the next chapter we show the research questions that meet our goals, and we intended to answer. They are divided into two groups: the first group is concerned with evaluating the proposed metrics as valid measures for code obfuscation security; the second group of questions investigate the effectiveness of code obfuscation using the proposed metrics.

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

- a researcher interested in empirical validation for code obfuscation;
- a software developer or project manager who wants to ensure a high resilience of obfuscated programs to deobfuscation attacks before delivering it to the customers [CCFB14].

We obfuscated 11 real-world applications of the 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, and the obfuscation took place on this level. In Section 7.3.5 we provide a comprehensive description of SPECjvm2008 suite, in Section 7.3.5 we provide an overview of SPECjvm2008's programs; the full documentation can be found on the benchmark's webpage [Spe08].

7.3 Experiment Planning

7.3.1 Tool-sets and Context

There are two kinds of variables in an experiment, independent and dependent variables [WRH+00]. The *independent variables* (predictor variables) are those variables that we can control and change in the experiment. We investigate the effect of the changes in the independent variables (proposed metrics) against the *dependent variable* (or outcome variable),[1] which is the security factor for obfuscated code (see Section 7.3.4). That is we aim to check if the independent variables reflect the level of security in obfuscated code using the dependent variable.

[1] Often there is only one dependent variable in an experiment.

7.3.2 Independent Variables Metrics

The set of independent variables, in this experiment, are based on selecting the following metrics for evaluation purposes.

Algorithmic Complexity Measures

We use four algorithmic complexity measures, which are proposed in Chapter 6:

Compression. Using compression we can approximate Kolmogorov complexity and then calculate the length of the compressed code using a normal compressor, see Section 2.9.

Normalised information distance. This metric is introduced in Definition 6.11. It measures the level of confusion (level of dissimilarity) in the obfuscated code. It is a normalised measure that takes a value between 0 and 1, where 0 indicates a complete similarity, meaning the obfuscation has not added any confusion to the clear code, and 1 means that the obfuscated code is totally dissimilar to the original code.

Unintelligibility Index. We present this measure in Definition 6.22. It aims to estimate how much more difficult to understand the obfuscated code in comparison to the original code, and to which extent code obfuscation transforms the complexity of the original code.

Normalised compression. This measure is proposed in Definition 6.25. It is an approximation of Kolmogorov complexity using compression, then normalised by dividing it with the upper bound of Kolmogorov complexity (size of the code); it is a non-negative number ranging from 0 to 1 and expresses the compressibility ratio of a certain code (in our case, a binary string).

Classical Complexity Measures

The purpose of including classical complexity measures are of two-fold. First, we need to compare our proposed measure to the most widely used complexity in code obfuscation. Secondly, we need to validate them to check if they are good candidates for measuring code obfuscation. All the classical complexity measures are computed using *Testwell CMTJava*, a commercial tool for Java code quality assurance [CMT15]. We will now provide an overview of the most commonly used classical complexity measures in code obfuscation.

MacCabe Cyclomatic complexity. McCabe Cyclomatic complexity [McC76] is a classical complexity measure, based on *Control Flow Graphs* (CFG). It measures the number of linearly-independent paths in the CFG of a program. McCabe's measure is an example of control flow metrics. The McCabe Cyclomatic complexity is computed as follows:

$$V(G) = e - n + 2 \cdot p$$

where $V(G)$ denotes the McCabe Cyclomatic complexity, e the number of edges in a graph, n the number of nodes in the graph, and p the number of connected components in the graph.

Halstead complexity measures. These metrics were proposed by Halstead [Hal77] to measure software complexity attributes based on what it called *software science*. Halstead's metrics are based on some program statistics that considers the source-code a sequence of tokens, and classifying them either as an operator or an operand token. These tokens are then counted and categorised as follows

- η_1 the number of unique operators.
- η_2 the number of unique operands.
- N_1 the total number of operators.
- N_2 the total number of operands.
- $\eta_1 + \eta_2$ the vocabulary of the program.

All Halstead's measures are derived from these four quantities with certain fixed formulas as described below.

Halstead Volume (*V*). A metric that describes the size of the programs implementation. It measures the information content of the program in bits;[2] it is calculated as the program length times the logarithm (base 2) of the program's vocabulary, such that:

$$V = (N_1 + N_2) \cdot \log_2(\eta_1 + \eta_2)$$

Halstead Difficulty (*D*). This metric describes the difficulty level of the program; it is

[2]The number of bits that are required to store a program of length N, provided that the operands and operators are encoded as binary strings of a uniform length.

proportional to the number of unique operators in the program.

$$D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$$

Halstead Effort (*E*). This metric describes the effort to implement or understand the program, it is proportional to the volume and the difficulty level of the program.

$$E = V \cdot D$$

Maintenance Index (*MI*). [CALO94] This is a composite metric that incorporates a number of traditional source-code metrics into a single number, formed by a linear regression equation (see Section 6.4.6 and Section 8.2.4), which indicates relative maintainability. *MI* comprises of weighted Halstead metrics (effort or volume), McCabes Cyclomatic Complexity, and the number of lines of code (*LOC*).

$$MI = 171 - 3.42 \cdot \ln(E) - 0.23 \cdot V(G) - 16.2 \cdot \ln(LOC)$$

7.3.3 Potency

We use the *Potency* (Pt_E) measure that is presented in Section 3.8; it measures how complex the obfuscated code in comparison to the original code, using classical complexity measures. We use this measure to study the effect of obfuscation using the above classical complexity before and after obfuscation. This measure is very similar to the unintelligibility measure (see Section 6.4.1).

7.3.4 Experimental Assumptions and Choice of Dependent Variable

As discussed in Chapter 5, the evaluation criteria for successful deobfuscation attacks are whether it is possible to produce the original code from the obfuscated version. The result of deobfuscation process has to satisfy, in addition to the functionality, proximity, and the amount of information needed by the deobfuscator, a set of properties that the original program satisfies. Determining these properties brings a challenge to the experiment. We assume that the main property which the attackers are interested in, and the defender is trying to secure, is the original code itself. Therefore, our success criteria for deobfuscation is to obtain the original code from the obfuscated one. In this

case, we can state that the deobfuscation is a successful attack. However, this is an extreme case and might sound unrealistic because the adversary may be satisfied with only partial information about the original code. To account for such conditions, we compute the percentage of successfully retrieving of any part of the original program from its obfuscated version. We can also assume that the attacker is hoping to construct a source-code, which is very similar to the original one, but not necessarily the same.

Normally determining whether obfuscated code is successfully deobfuscated (in our case decompiled), requires a human subject to check. However, involving human subjects brings a lot of challenges in terms of time and cost. To alleviate these challenges, we turn into an alternative method. Ultimately, as we discussed in Chapter 4, the aim of obfuscation is to protect the clear code by obscuring its implementation, we argued this point in Chapter 5, so an attacker should not easily retrieve that code. We made an assumption, here, we take advantage of the presence of unobfuscated source-code in our hands: we assume that the unobfuscated code is fairly easy to attack (reverse) comparing to the obfuscated code. Therefore, the decompiled code (deobfuscated) that is close to (or matches) the unobfuscated code, will provide an indication of the effectiveness of the decompilers at producing an easily understandable code. Hence, we use this assumption to obtain the dependent variable: the percentage of code obfuscation resilience to decompilation.

Dependent Variable

Now the question is how to establish a method to compare the decompiled code with the original clear code without involving human subjects, i.e. aiming to reduce the effect of human subjects in obfuscation process, we hold constant the individuals that perform deobfuscation; so to determine the level of the attack result, we compare the decompiled obfuscated source-code to the original clear source-code using a plagiarism detection tool. The role of the plagiarism detection tool, here, is to measure the percentage of retrieved unobfuscated clear code in the decompiled code (deobfuscated code). For convenience, we report the percentage of dissimilarity of decompiled code (deobfuscated) to the original one. We use a plagiarism detection and file comparison tool (diff) called *Sherlock* [she15] which is used as a part of BOSS online submission system at Warwick university. It aids in estimating the percentage of inferred clear code from the obfuscated decompiled `jar` file. The dependent variable that we want to validate against is the percentage of retrieved code after conducting a decompilation attack. We discuss the choice of threat model in Section 7.3.7.

7.3.5 Context: Benchmark

We used SPECjvm2008 (Java Virtual Machine Benchmark) programs as a subject of our experiment and to evaluate the proposed metrics. We applied 11 real-world applications of the benchmark suite, containing several real life applications and benchmarks focusing on core Java functionality.[3] Each one was written in Java source language and compiled with `javac` to Java byte-code; the obfuscation took place on this level. The following is a brief description of each program in the benchmark as provided by the SPECjvm2008 documentation webpage.

Check. A program that checks JVM and Java features and tests if the Java platform is suitable to run the benchmarks.

Compiler. This is a Java decompiler that uses the OpenJDK (JDK 7 alpha) front-end compiler to compile .java files. The code compiled is `javac` itself and the sunflow sub-benchmark from SPECjvm2008.

Compress. Compresses data, using a modified Lempel-Ziv method (LZW). It basically finds common substrings and replaces them with a variable size code. This is deterministic, and can be done on the fly.

Crypto. It provides three different ciphers (AES, RSA, and signverify) to encrypt data:

- **aes.** Encrypt and decrypt using the AES and DES protocols, using CBC/PKCS5Padding and CBC/NoPadding.
- **rsa.** Encrypt and decrypt using the RSA protocol.
- **signverify.** Sign and verify using MD5withRSA, SHA1withRSA, SHA1withDSA and SHA256withRSA protocols.

Derby. An open-source database written in pure Java. The focus of this benchmark is on BigDecimal computations (based on the telco benchmark) and database logic, especially on locks behaviour.

MPEGaudio. MPEG-3 audio stream decoder, from Fraunhofer Institut fuer Integrierte Schaltungen. Its mp3 library leverage JLayer, an LGPL mp3 library, with heavy floating-point calculations and a good test of mp3 decoding.

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

Scimark. This benchmark was developed by NIST and is widely used by the industry as a floating point benchmark. It consists of five subtest (fft, lu, monte-carlo, sor, sparse).

Serial. This benchmark serializes and deserializes primitives and objects, using data from the JBoss benchmark. The benchmark has a producer-consumer scenario where serialized objects are sent via sockets and deserialized by a consumer on the same system.

Startup. This benchmark starts each benchmark for one operation. A new JVM is launched and time is measured from start to end. The start-up benchmark is single-threaded. This allows multi-threaded JVM optimizations at start-up time.

Sunflow. Tests graphics visualization using an open source, internally multi-threaded global illumination rendering system. The sunflow library is threaded internally, i.e. it is possible to run several bundles of dependent threads to render an image.

XML. This benchmark has two sub-benchmarks: XML.transform and XML.validation. XML.transform exercises the JRE's implementation of javax.xml.transform (and associated APIs) by applying style sheets (.xsl files) to XML documents. XML.validation exercises the JRE's implementation of javax.xml.validation (and associated APIs) by validating XML instance documents against XML schemata (.xsd files).

7.3.6 Obfuscators

Two obfuscators were used: one commercial *DashO* [das15] *evaluation copy* and a free source version of *SandMark* [san15]. The original benchmarks `jar` files were obfuscated by using 44 different obfuscation techniques of the *DashO* and *Sandmark* obfuscators.

Sandmark. A state-of-the-art code obfuscation suite developed at the University of Arizona. It has an advantage over other available open-source obfuscation suites for Java byte-code; despite being around for a long time, it has a high level of flexibility, customisation with a large number of obfuscation techniques. Besides obfuscation, it provides a list of watermarking algorithms, a set of static code analyzers, performance evaluation, a set of software engineering metrics and other static code statistics.

All the obfuscation techniques in Sandmark are clustered into three groups: application, class, and method obfuscation levels. Application level obfuscation techniques apply obfuscation over

all the program's components, where method and class obfuscations allow users to specify which method or class to choose for obfuscation i.e. it provides a fine-grained flexibility over application level obfuscation. This way Sandmark can deliver obfuscation for only essential parts of targeted programs, which helps to improve the performance of obfuscated program, as obfuscation, in most cases, penalises the performance. The complete list of Sandmark obfuscations and their description is provided in Table 7.3, 7.2 and 7.4 .

Dasho. The second obfuscation tool, Dasho, is a closed-source commercial obfuscator tool developed by PreEmptive Solutions. It provides obfuscation for DotNet, Java and Android applications including tamper notification and detection techniques. It uses obfuscation algorithms based on control flow obfuscation, patented overload induction methods for renaming and code encryption; in addition to exception handling obfuscations, optimisation and compacting techniques. Dasho has an advantage among commercial and open-source obfuscators; it indirectly applies optimisation techniques, which reduces the size of the code, making the obfuscated code smaller and faster to run. The complete list of Dasho obfuscations and their description is provided in Table 7.1.

Technique	**Type**	**Abbr**	**Description**
ControlFlow	Control-Flow	DH-C-CF	This process synthesizes branching, conditional, and iterative constructs that produce valid forward (executable) logic, but yield non-deterministic semantic results and produces spaghetti logic when decompilation is attempted.
Optimisation	Data	DH-D-OP	Performs algebraic identity, strength reduction, Constant folding and other peephole optimizations.

Continued on next page

Technique	Type	Abbr	Description
Synthetic	Data	DH-D-SY	This is an obfuscation technique that marks methods and fields as a special 'synthetic', generated by the Java compiler that links a local inner class to a block's local variable or reference type parameter. It helps confusing some decompilers.
Rename FlattenHierarchy OverInduction	Control-Flow	DH-C-RFO	A patented algorithm devised by PreEmptive Solutions. Overload Induction will rename as many methods as possible to the same name, in addition to flattening the structure of package hierarchy by putting all the renamed classes into the default package.
Rename FlattenHierarchy Simple	Layout	DH-L-RFS	Assigns a random name to each identifier in the program, where all the renamed classes are put into the default package
Rename OverInduction Maintain-hierarchy	Layout	DH-L-ROM	Similar to Rename FlattenHierarchy OverInduction; however, the package naming hierarchy is retained.
TryCatch10	Control-Flow	DH-C-TC	Try/Catch handlers are added to methods to further confuse decompilers, on a scale of 1-10.

Table 7.1: Dasho obfuscation techniques.

Technique	Type	Abbr	Description
Constant Pool Reordering	Layout	SM-L-CPR	Reorders the constants in the constant-pool and assigns random indices to them. The constants are randomly assigned some unique index within the length of constantpool.
Objectify	Layout	SM-L-OB	Replaces all the fields in a class with fields of the same name with a type Object.
Rename Register	Layout	SM-L-RR	Renames local variables to random identifiers.

Table 7.2: Sandmark Layout obfuscation techniques

Technique	Type	Abbr	Description
Array Folding	Data	SM-D-AF	Folds one-dimensional array into a multi-dimensional array.
Block Marker	Data	SM-D-BM	Used to hide a watermark and diversify the byte-code, by randomly marking all basic byte-code blocks in the program with either 0 or 1.
Bludgeon Signatures	Data	SM-D-BS	Confusing the signatures of methods, by making all of the static method signatures the same, and convert all the parameters to the type `object`.
False Refactor	Data	SM-D-FR	It can be applied on two classes that have no common behaviour. 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.

Continued on next page

Technique	Type	Abbr	Description
Field Assignment	Data	SM-D-FA	Inserting a bogus field into a class and then making assignments to that field in specific locations throughout the code.
Integer Array Splitter	Data	SM-D-AS	Splits a single array of integers into two arrays and modifies all the array initialization, read, write, and arraylength references consistently of the two arrays.
Merge Local Integers	Data	SM-D-MLI	Combines two `int` variables into a single long variable.
Overload Names	Data	SM-D-ON	Change methods names to similar names. Method overriding relationships remain intact, whereas existing overloaded methods may be destroyed, and the new ones created.
Param Alias	Data	SM-D-PA	Tries to find a (non-initializer, non-abstract, non-native) method in a class that takes some object type as a parameter. It then aliases that parameter within the method using Thread Local Storage (the ThreadLocal class). Every load of the parameter is replaced with ThreadLocal.get(), and every store is replaced with `ThreadLocal.set` (Object).
Promote Primitive Register	Data	SM-D-PPR	Replaces all the local `int` variables in a function with local `java.lang.Integer`.
Promote Primitive Types	Data	SM-D-PPT	Changes all primitives in every method into instances of the respective wrapper classes.

Continued on next page

Technique	Type	Abbr	Description
Publicise Fields	Data	SM-D-PF	Makes the fields of a class public.
Reorder Parameters	Data	SM-D-RP	An obfuscator that shuffles the argument orders for all methods.
Static Method Bodies	Data	SM-D-SMB	Static Method Bodies splits all of the nonstatic methods into a static helper method and a nonstatic stub that calls it.
String Encoder	Data	SM-D-SE	It obfuscates the literal strings of a program. Each string is 'encrypted' and any string reference is replaced by a call to a method that 'decrypts' it.
Variable Reassigner	Data	SM-D-VR	Reallocates the local variables in a method, in order to try to minimize the number of local variable slots used.
Duplicate Register	Layout	SM-D-DR	This algorithm creates an additional variable that has its value changed in coordination with an original local variable. Each reference to that variable value may have been changed to reference the new variable instead.

Table 7.3: Sandmark Data obfuscation techniques

Technique	Type	Abbr	Description
Branch Inverter	Control-Flow	SM-C-BI	Exchanges the `if` and the `else` part of an `if-else` statement. It also negates the `if` instruction, for example `IFGE` JVM instruction is followed by `IFLT` another JVM instruction which negates the effect of `if`, so that the semantics is preserved.

Continued on next page

Technique	Type	Abbr	Description
Buggy Code	Control-Flow	SM-C-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.
Class Splitter	Control-Flow	SM-C-CS	Adds several spurious classes by splitting the original, non-obfuscated code, into several obfuscated ones.
Duplicate Register	Control-Flow	SM-D-DR	This algorithm creates an additional variable that has its value changed in coordination with an original local variable. Each reference to that variable value will be changed to reference the new variable instead.
Dynamic Inliner	Control-Flow	SM-C-DI	Dynamically inlines (replacing method invocations) method bodies at the runtime using `instanceof`.
Inliner	Control-Flow	SM-C-IN	Inlines static method bodies by replacing method invocations.
Irreducibility	Control-Flow	SM-C-IR	Adds conditional branches to a method via opaque predicates so that the control flow graph of the resulting method is irreducible.

Continued on next page

Technique	Type	Abbr	Description
Method Merger	Control-Flow	SM-C-MM	Merges all of the public static methods that have the same signature in each class into one large master method.
Opaque Branch Insertion	Control-Flow	SM-C-OBI	Randomly inserts branches into a method using a library of opaque predicates.
Random Dead Code	Control-Flow	SM-C-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.
Simple Opaque Predicate	Control-Flow	SM-C-SOP	Implements simple boolean opaque predicate (see Section 3.4.2) and adds them to the user's code. The aim is to embed opaquely true constructs which should be stealthy.
Split Classes	Control-Flow	SM-L-SC	This is a Node Splitting obfuscation algorithm which obfuscates a class file by splitting a node into two. 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.

Continued on next page

Technique	Type	Abbr	Description
Transparent Branch Insertion	Control-Flow	SM-C-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.

Table 7.4: Sandmark Control-Flow obfuscation techniques

Figure 7.1: Schematic overview of how we applied decompilation and the metrics in the experiment.

7.3.7 Decompilation as Threat Model

In this experiment we evaluate the resilience of obfuscation algorithms by using the static analysis techniques leveraged in decompilers. The purpose of decompilation, in this experiment, is to produce code that is easy to comprehend i.e. to be able to understand the program. The strict syntactic correctness is not fully required, as partially decompiled code may be sufficient for understanding. We do not expected decompilers to produce a perfect code or a source-code exactly similar to the original code. Dagenais and Hendren in [DH08] demonstrated the possibility of constructing Java code from partially correct decompiled Java code.

Java decompilation process. Decompilation can be used as a reverse-engineering tool in Java byte-code, which consists of low-level stack-based byte-code instructions; the decompiler transforms these low level instructions to high-level Java source-code. Generally the task of decompiling Java byte-code is relatively simpler than other binary code decompilers, as Java byte-code contains extensive meta-data, such as method names and their signature, class names and types, which makes the task of decompilation much easier. More specifically, the Java decompilation process involves the following analysis phases:

- **Local variable typing inference.** Java byte-code generally preserves the type information for fields, method returns and parameters, however it does not have type information for local variables.
- **Merging stack-based instructions into expressions and variables.** Stack variables in Java byte-code, are mostly due to the optimisation process; they are kept on the stack to enhance byte-code performance. It requires identifying such variables by decompilers in order to translate them to local variables and expressions.
- **Arbitrary control flow construction analysis.** This is the process of reconstructing the unstructured and arbitrary control flow of byte-code into readable high level source-code.
- **Exceptions and synchronisation handling.** This recovers all the exception handlers and `synchronized()` statements from the Java byte-code instruction and the meta-data.

We evaluate our proposed metrics using three Java decompilers, *JD* [JD15], *JAD* [JAD15] and *Jode* [Jod15], to investigate the resilience, and assess to which extend the applied obfuscation techniques can resist decompilation attacks. Our choice was based on a study by Hamilton and Danicic [HD09], who investigated the effectiveness of Java decompilers using an empirical evaluation on a group of currently available Java byte-code decompilers. We selected, based on that experiment, three Java decompilers that score the best among all decompilers in terms of effectiveness and correctness, *JD, JAD* and *Jode*.

JD. (Java Decompiler) A freeware Java decompiler, provided as a GUI tool and command-line version, as well as in the form of plug-ins for the Eclipse (JD-Eclipse) and IntelliJ IDEA (JD-IntelliJ). It targets Java 5 byte-code and later versions.

JAD. A freeware and a popular decompiler for non-commercial use, with no source-code available, and is no longer maintained. It is written in C++, and is relatively fast compared to other Java decompliers. Jad is used as the back-end by many decompiler GUIs including an Eclipse IDE plug-in (JadClipse).

Jode. (Java optimise and decompile environment) is an open source decompiler and obfuscator/optimiser. *Jode* has a verifier, in a similar way to the Java runtime verifier, which tries to find type data from byte-code class files. *Jode* is able to correctly infer types of local variables, and is able to transform code into a more readable format, closer to the way Java code is naturally written.

7.3.8 Choice of Compressor

Most of the available compressors are 'Normal' according to [CV05] (see Section 2.9), and subsequently any of them can be used to approximate the proposed metrics. However, we had to make the choice amongst the most effective and accurate ones. Better compression which has a high *compression rate*[4] is an important factor to have an effective approximation for Kolmogorov complexity, but this statement is not always true for *NCD* [CV05]. In a study by Cebrián et el. [CAO05], using three different compressors *bzip2, gzip*, and *PPMZ*, *NCD* did not satisfy the idempotent property (see Definition 2.39) in some cases, it shows that the compression skewed according to the size of measured file. However, *PPMZ* showed more resilience comparing to the rest of compressors with an average error of 0.1043%. A most recent study by Alshahwan et al. [ABCD15], in the context of malware detection using *NCD*, a comparison was made among three compressor *winzip, gzip,* and *7zip (PPMZ)* using the same testing technique as in [CAO05]. *7zip* performed better than the other compressors. It scores an idempotent result close to zero in most files size, with a window size of a maximum *4GB*. Based on these two studies we decided to use *PPMZ* for the experiment, which is implemented as a python library.

7.4 Experimental Process

For each version of a `jar` file obfuscated under a certain obfuscation technique, *NCD*, *NC*, and π_U were calculated. All the obfuscated programs are subjected to an automatic decompilation process.

[4] Data compression rate is a ratio that is calculated by dividing the uncompressed size of a (binary) program or file by its compression size.

Additionally, we applied the decompilation process to the clear original programs, which helps in assessing the difference between the original source-code and obfuscated source-code decompiled using the same decompiler. The difference was calculated by computing the percentage of original code extracted from the deobfuscated one using *Sherlock* (see Section 7.3.4). The code matching (using *Sherlock*) was calculated between the decompiled `jar` file and the original file, to estimate the percentage of the retrieved code from the decompiled `jar` file.

We build a python tool to compute the proposed metrics (NCD, π_U, NC, C). We automate the whole testing process, using a python script to glue the command line versions of Sandmark, Dasho obfuscation, decompilation, classical complexity measures and the tool that compute the proposed metrics. The results were saved in a repository (`.mat`), and all data analysis and statistical testing were conducted in Matlab [Mat15]. All of the above components are integrated into our prototype, Fig. 7.2 shows an overview of the tool-sets and the experimental procedure.

Figure 7.2: High level overview of the experimental procedure

7.5 Analysis Methodology

We apply statistical methods to verify and analyse our results. Descriptive statistics are applied in order to present the numerical processing of a data set, and to graphically present important aspects of the experiment data set (the collected numerical data results) and how it is distributed.

We use three types of descriptive statistics: measures of central tendency, dependency, and

graphical visualization. In measures of central tendency we use the *mean* which indicates the *middle* of a data set. This *midpoint* is normally called the *average* and is interpreted as an estimation of the expectation of the stochastic variable from which the data sets are sampled [She07]. The mean of a given sample of data points $x_1 ... x_n$, is denoted by $\tilde{x}$:

$$\tilde{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

To measure the dependency between variables we used two types of statistical measurements: correlation and regression analysis. Correlation is a number that quantifies how much two data sets vary. We use the *Spearman rank correlation coefficient* to express this relation.

Spearman rank correlation ρ [She07]. This is a ranked version of Pearson's correlation coefficient and belongs to non-parametric statistics, denoted by ρ. Non-parametric statistics do not make any assumptions about the data distribution from which the sample was drawn [She07]. The Spearman rank correlation is used as an effect size, and estimates the magnitude of effect or association between two or more variables [SL93] when paired quantitative data are available; it helps to relate dependent variables (Decompilers efficiency) to independent variables (*NCD* and *NC*). Given two sets of data samples, $x_1 ... x_n$ and $y_1 ... y_n$.

$$\rho = \frac{\sum_{i=1}^{n}(x_i - \tilde{x})(y_i - \tilde{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \tilde{x})^2 (y_i - \tilde{y})^2}}$$

Spearman's ρ can vary in magnitude from -1 to 1, with -1 indicating a perfect negative linear relation, 1 indicating a perfect positive linear relation, and 0 indicating no linear relation between two variables.

To further assess the relationship among the different variables (decompilation metrics), a regression model was investigated as another method to check for dependency.

Regression analysis. This is a statistical technique for studying the relationships between the independent and dependent variables (see Section 7.3.4 and Section 7.3.2). It provides a method to predict changes of dependent variable(s) by looking at independent variable(s). The outcome of regression analysis is normally a regression equation (see also Section 6.4.6). The main difference between a regression analysis and a correlation coefficient is that regression looks for

prediction, whereas the correlation coefficient only compares the level of dependency of two variables. However, they are strongly related as higher correlation among variables indicates more accurate and precise prediction of the dependent variable from independent variables. The linear regression method is used to construct the regression equations. The coefficient of Determination (R^2) [She07] value is used as a guideline to measure the accuracy of the data model, i.e. how well data points fit a statistical model.

Hypothesis testing. A statistical hypothesis test is a form of statistical inference, which allows one to investigate evidence that supports some claims based on data taken from controlled experiment or an observational study [She07]. The methods of inference used to support or reject claims based on sample data are known as *Tests of significance*. Tests of significance are conducted by applying what is called the *null hypothesis*, which represents a theory or a general statement that has been put forward, either that there is no relationship between two measured variables, or it is to be used as a basis for argument, but has not been proved. The null hypothesis describes some properties of the distribution from which the sample is drawn; the aim is to reject that these properties are true with a given statistical significance. Therefore, the null hypothesis should be formulated negatively. An alternative hypothesis is the hypothesis that is chosen when the null hypothesis is rejected. The aim of hypothesis testing is to investigate if it is possible to reject a certain null hypothesis H_0.

For hypotheses validation, we applied *p-values* drawn from Spearman rank correlation ρ, and *Mann-Whitney test* [WRH+00], which is a non-parametric alternative to the *t-test* [WRH+00], as we do not make any assumption about the normality of distributions in the test samples. The p-value is used to measure the probability of the statistical significance of evidence under test i.e. representing the probability that obtained test results are due to a Type I error :

$$\Pr(\text{type I error}) = \Pr(\text{Reject } H_0 \,|\, H_0 \text{ true})$$

This can be explained as the probability of falsely rejecting H_0, the null hypothesis. This test allows for checking the presence of significant differences in the paired data, and hence to weigh the strength of the evidence. The null hypothesis is rejected if the calculated p-value turns out to be less than a predetermined significance level (α). In this experiment, the p-value should be less

than 0.05 to reject the null hypothesis [She07]. In all of the statistical tests, we consider 95% as a significance level, i.e. we accept 5% probability of committing a type I error.

In the *Mann-Whitney test*, similarly to the Spearman rank correlation, the p-value threshold is set at 0.05 to reject the null hypothesis. Given two independent samples: $x_1, ..., x_n$ and $y_1, ..., y_m$. We rank all samples and calculate U_w.

$$U_w = N_A \cdot N_B - \frac{N_A \cdot (N_B + 1)}{2} - T \text{ and } U'_w = N_A \cdot N_B - U_w$$

where $N_A = \min(n; m), N_B = \max(n, m)$, and T is the sum of the ranks of the smallest sample. So we can reject the null hypothesis if $\min(U_w, U'_w)$ is less than or equal to a criterion value [WRH+00].

7.6 Summary

We provide a detailed description of the key elements in the design of an experiment, for the purpose of evaluating a set of proposed metrics for measuring the quality of code obfuscation. The aim of this experiment is to complement the previous theoretical results, to provide empirical evidence, and to show the usefulness of algorithm information theoretical metrics reflecting the quality and security in obfuscated code.

We based the experimentation design and methodology of analysis on Wohline et al.'s framework on experimental software engineering. A comprehensive description of the tool-sets and experimental context are given, which includes two obfuscators: Sandmark, an open source suite, and Dasho, a commercial tool, applied to SPECjvm2008 suite. We choose three different decompilers (*JD, JAD, and Jode*) as the model of the adversary.

8 Experimental Results and Analysis

In this chapter we report the outcome of our experiments, and state formally the research questions and formulate their null hypotheses; then we provide the analysis and interpretations of the obtained results. We test the null hypotheses and evaluate the relation between our proposed metrics, the classical complexity metrics and the percentage of code obfuscation resilience to decompilation attacks, by providing evidence based on statistical analysis. We also use the proposed metrics to analyse and study the effect of code obfuscation techniques on programs, and quantify their resilience to decompilers.

8.1 Introduction

In the previous chapters we provided all the design requirements to set up, implement, and conduct experiments on a set of obfuscated SPECjvm2008 programs. In this chapter we proceed by stating the research questions, and their null hypotheses. Then we conduct a statistical analysis, which investigates the null hypothesis for each of the posed question by providing empirical validations based on Spearman rank correlation and Mann-Whitney test. We first check if the proposed metrics, normalised compression distance (*NCD*), unintelligibility Index (π_U) and normalised compression (*NC*), are measuring the code obfuscation resilience. Secondly, we compare the metrics with other classical complexity metrics, Cyclomatic complexity (*V(G)*), Halstead Difficulty (*D*), Halstead Effort (*E*), and Maintenance Index (*MI*). Then we apply the validated metrics to answer the rest of the questions. In particular, we examine if the obfuscation techniques produce any changes in the Kolmogorov complexity (measured by compression), and whether these changes are positive, i.e. if obfuscation techniques increase the complexity of the clear unobfuscated programs. Additionally, we investigate if the obfuscation process produces any changes in the unobfuscated programs of the SPECjvm2008. We also fit a multi-linear regression model aiming to produce a single

quantitative value on a percentage scale, which predicts the resilience of Java code obfuscation against decompilers. We proceed to provide a deep analysis of Dasho and Sandmark's obfuscation techniques using the proposed metrics, followed by studying the impact of decompilers on each obfuscation technique.

The overall results shows that our proposed metrics are empirically valid to measure the quality of code obfuscation. These metrics outperform the classical complexity measures in terms of being correlated with the degree of code obfuscation's resilience to decompilers. However, there is only one exception that is related to *NC*: this measure shows a weak correlation comparing to the other metrics involved in the study. The outcome of the analysis of the results sheds a light on the importance of taking into account the attack model when measuring the quality of code obfuscation. Applying any quantitative measure without parametrising it to a specific attacker can be misleading, and creates a false sense of security. We show that by comparing the values of the metrics of each obfuscated program, before and after decompilation attacks.

The remainder of this chapter is structured as follows. In Section 8.2, we present the research questions, their formulated null hypotheses, and provide the validation results of the proposed metrics model, using statistical hypotheses testing. Section 8.2.3 provides the statistical comparison with the classical complexity metrics. In Section 8.3.2 we analyse the impact of code obfuscation using the proposed metrics. Section 8.4 presents the potential threats and limitations that may affect the validity of the experimental results.

8.2 The Validation Results for the Proposed Model

In this section we will empirically verify whether the proposed measures: (*NCD*, π_U and *NC*) are valid metrics to quantify the security in code obfuscation. Specifically, we try to answer the following two questions:

($\mathbf{RQ}_1$) Do the proposed metrics (*NCD*, π_U and *NC*) reflect the amount of confusion added due to the obfuscation process?

($\mathbf{RQ}_2$) Do the classical complexity metrics reflect the same amount of confusion as the proposed metrics?

8.2.1 Hypotheses Formulation

We formulate the statistical hypothesis for each of the aforementioned research questions. An important aspect of the experiment is to formally state clearly what is going to be evaluated in the experiment, which leads to the formulation of the hypotheses below.

For each of the research questions, we formulate the subsequent null hypotheses groups to be tested: For the first research question **RQ1** we established the following null hypotheses:

- Null hypothesis, $\mathbf{H}_{10a}$: *NCD does not* measure the decrease in the efficiency of an attacker using *JD, JAD* and *Jode* decompilers, trying to construct a Java source-code similar to the decompiled original code.

  Alternative hypothesis, $\mathbf{H}_{11a}$ (not $\mathbf{H}_{10a}$): *NCD does* measure the decrease in the efficiency of an attacker using *JD, JAD* and *Jode* decompilers, trying to construct a Java source-code similar to the decompiled original code.

- Null hypothesis, $\mathbf{H}_{01b}$: *NC does not* measure the decrease in the efficiency of an attacker using *JD, Jad* and *Jode* decompilers, trying to construct a Java source-code similar to the decompiled original code.

  Alternative hypothesis, $\mathbf{H}_{11b}$ (not $\mathbf{H}_{01b}$): *NC does* measure the decrease in the efficiency of an attacker using *JD, JAD* and *Jode* decompilers, trying to construct a Java source-code similar to the decompiled original code.

- Null hypothesis, $\mathbf{H}_{01c}$: π_U *does not* measure the decrease in the efficiency of an attacker using *JD, JAD* and *Jode* decompilers, trying to construct a Java source-code similar to the decompiled original code.

  Alternative hypothesis, $\mathbf{H}_{11c}$ (not $\mathbf{H}_{01c}$): π_U *does* measure the decrease in the efficiency of an attacker using *JD, JAD* and *Jode* decompilers, trying to construct a Java source-code similar to the decompiled original code.

Effectively, there are nine null hypotheses that can be constructed from the above hypotheses, i.e. three null hypotheses for *NCD against JD, Jode and JAD decompilers*, and similarly three null hypotheses for NC and π_U respectively. For convenience, we grouped them into one main hypotheses according to the used measure. The above hypotheses are *one tailed* [She07], also known

as a *directional hypothesis*, which is a test of significance to determine if there is a relationship between the variables in one direction. We apply one-tailed hypotheses testing since we are only interested in one direction of analysing the validity of the proposed measure against the resistance to attacks using decompilation.

The null hypotheses suggest the presence of one main dependent variable, the *resilience variable* that measures the efficiency resisting decompilation (see Section 7.3.4). The independent variables involved in this experiment are the *NCD*, π_U and *NC* values.

For the second question **RQ2**, we construct the following null hypothesis:

- Null hypothesis, $\mathbf{H}_{02}$: The proposed measures *do not* perform better than the classical complexity metrics in measuring the quality of code obfuscation against decompilers attacks.

  Alternative hypothesis, $\mathbf{H}_{12}$ (not $\mathbf{H}_{02}$): The proposed measures *do* perform better than the classical complexity metrics in measuring the quality of code obfuscation against decompilers attacks.

We will start by investigating the question in **RQ1** and its formulated hypotheses in Section 8.2.1, then we will address **RQ2**, which shows how well the classical complexity metrics are performing comparing to our proposed model.

The non-negative unintelligibility index π_U indicates a higher complexity of obfuscated code and less code comprehensibility (unintelligible) with respect to the original benchmark files, where a negative value indicates less potent code (see Section 6.4.1). *NCD* reports the dissimilarity between obfuscated programs and the original clear code. *NCD* values are ranged from 0 (exact similarity), which means the obfuscation did not add any confusion to the original code, and 1 (totally dissimilar) which indicates the maximum level of confusion added to the original code (see Theorem 6.27). Similarly, *NC* reports values between 0 and 1; *NC* = 0 for empty strings, this value cannot to occur in this experiment, as all the programs are non-empty. *NC* = 1 indicates the highest complexity of obfuscated code that can be achieved with respect to the maximum degree of compression of obfuscated code (see Section 6.4.2).

Each of the proposed metrics is applied to each obfuscated program of the SPECjvm2008; then we check if this metric is correlated with the amount of retrieved code due to the different decompilers, i.e. the degree of code obfuscation resilience (see Section 7.3.4).

8.2.2 Validation Analysis using Correlation and Hypothesis Testing

In this section we investigate the association and the linearity between *NCD*, π_U and *NC* and the degree of resilience of obfuscated programs, i.e. the average percentage of failure to retrieve the original code from decompiled obfuscated code; this value is denoted, in the reported results (tables and graphs), by a notation G_{decomp}, according to the name of used decompiler, or G_{AllDec}, which refers to all decompilers involved in this experiment [1] (see Fig. 7.1).

We test this relation by relying on statistical methods such the Spearman rank coefficient correlation (see Section 7.5). We calculate the Spearman rank coefficient correlation ρ between *NCD*, π_U, and *NC* and the resilience of obfuscated programs in Fig. 8.1: each entry in the tables (the figure) corresponds to the Spearman rank coefficient correlation ρ between the proposed metrics (row) and the percentage of resilience to decompilers (column); for example, the entry (NCD_{JAD} versus G_{JAD}) of Fig. 8.1-(a), which is equal to 0.88, corresponds to the correlation between *NCD* measure after decompiling the obfuscated programs (the benchmark) with *JAD* and the percentage of resilience to *JAD* (G_{JAD}). The overall correlation between *NCD* and all inferred programs (due to decompilation) achieves a result close to 0.89 (strong correlation). Similarly, we can notice the same correlation with respect to each decompiler. In the case of the π_U measure, the Spearman rank coefficient correlation ρ scores an overall result of around 0.48 (moderate correlation). On the other hand, *NC* obtains a weak correlation of around 0.11. We further check for the correlation in the case of simple compression (C);[2] we find also a positive correlation, surprisingly (see Section 6.4.2), performing much better than *NC*.

	G_{JAD}
NCD_{JAD}	0.88
$\pi_{U_{JAD}}$	0.51
NC_{JAD}	0.23
C_{JAD}	0.24

(a)

	G_{JD}
NCD_{JD}	0.90
$\pi_{U_{JD}}$	0.48
NC_{JD}	0.14
C_{JD}	0.32

(b)

	G_{Jode}
NCD_{Jode}	0.85
$\pi_{U_{Jode}}$	0.39
NC_{Jode}	0.18
C	0.25

(c)

	G_{AllDec}
NCD_{AllDec}	0.89
$\pi_{U_{AllDec}}$	0.48
NC_{AllDec}	0.11
C_{AllDec}	0.22

(d)

Figure 8.1: The Spearman rank coefficient correlation ρ between the proposed metrics and the resilience of obfuscated programs (percentage of the clear code that was not recovered) using decompilers (*JD, JAD*, and *Jode*).

In general, the correlations are positive; this indicates a strong positive direction of association.

[1] For convenience and the ease of reading, we indicate the inferred program from the deobfuscated code with *G* subscript with the name of decompiler G_{JAD}, G_{JD} and G_{Jode}, i.e. the average retrieved data are due to decompiling an obfuscated code.

[2] *C* is used as the main building block in the all of the proposed metrics.

NCD tends to increase as G_{AllDec} increases. This can, indeed, provide evidence that *NCD* can measure the impact of code obfuscation resilience to attacks (decompilers). Similarly, π_U and *NC* show a positive correlation; however, much lower than the *NCD* correlation.

We also perform a significance test to decide whether based upon the experimental results, there is any or no evidence to suggest the existence of a linear correlation between the proposed metrics (*NCD*, π_U, *NC*) and G_{AllDec}, in general, and according to each decompiler, $G_{JAD}, G_{JD}, G_{Jode}$. To achieve this, we test the null hypothesis (see Section 8.2.1) $\mathbf{H}_{10a}$ against the alternative (research) hypothesis $\mathbf{H}_{11a}$.

	G_{JAD}
NCD_{JAD}	$\ll 0.05$
$\pi_{U_{JAD}}$	$\ll 0.05$
NC_{JAD}	$\ll 0.05$
C_{JAD}	$\ll 0.05$

(a)

	G_{JD}
NCD_{JD}	$\ll 0.05$
$\pi_{U_{JD}}$	$\ll 0.05$
NC_{JD}	0.008
C_{JD}	$\ll 0.05$

(b)

	G_{Jode}
NCD_{Jode}	$\ll 0.05$
$\pi_{U_{Jode}}$	$\ll 0.05$
NC_{Jode}	$\ll 0.05$
C_{Jode}	$\ll 0.05$

(c)

	G_{AllDec}
NCD_{AllDec}	$\ll 0.05$
$\pi_{U_{AllDec}}$	$\ll 0.05$
NC_{AllDec}	$\ll 0.05$
C_{AllDec}	$\ll 0.05$

(d)

Figure 8.2: The p-values of the Spearman rank coefficient correlation ρ between the proposed metrics and the resilience of obfuscated programs using decompilers (*JD, JAD*, and *Jode*).

We conduct the hypothesis testing by computing the p-value (see Section 7.5) in Fig. 8.2: each entry in the tables (the figure) shows the p-value of testing a null hypothesis; for example, in Fig. 8.2-(a) the p-value of testing the null hypothesis of whether the value ofNC_{AllDec}, i.e. NC is applied to the decompiled version of the obfuscated programs (the benchmark) with all decompilers, correlates with G_{AllDec} (NC_{AllDec} versus G_{AllDec}) is 0.008, which is significant because it is lower than 0.05 threshold.

All the p-values that are used to test $\mathbf{H}_{10a}$, (*NCD*, π_U, *NC*) versus ($G_{AllDec}, G_{JAD}, G_{JD}, G_{Jode}$), are below the significant levels, i.e. p-value < 0.05, as we see in Fig. 8.2. Therefore, we can reject the null hypotheses of $\mathbf{H}_{10a}$ group (see Section 7.5) and accept the alternative hypotheses $\mathbf{H}_{11b}$ group. This implies that the correlation of (*NCD*, π_U, *NC*) versus ($G_{AllDec}, G_{JAD}, G_{JD}$, and G_{Jode}) are indeed significant.

8.2.3 Comparison with Classical Metrics

In this section we try to answer the second question **RQ2**, which is about comparing classical complexity metrics with our proposed metrics, namely *NCD*, π_U and *NC*, in addition to the simple

compression length (*C*).[3] We answer this question by investigating the null hypothesis in $\mathbf{H}_{02}$ using the same analysis methodology as with the proposed metrics. We have five different classical complexity measures: Cyclomatic complexity (*V(G)*), Halstead Difficulty (*D*), Halstead Effort (*E*), and Maintenance Index (*MI*); all these measures are discussed in Section 7.3.2. We investigate the potency (see Section 3.8) of each of these metrics for the same purpose of comparison with the proposed metrics.

	G_{JAD}
LOC_{JAD}	0.25
$V(G)_{JAD}$	0.21
D_{JAD}	0.18
E_{JAD}	0.22
MI_{JAD}	0.19

(a)

	G_{JD}
LOC_{JD}	0.28
$V(G)_{JD}$	0.26
D_{JD}	0.14
E_{JD}	0.13
MI_{JD}	0.31

(b)

	G_{Jode}
LOC_{Jode}	0.04
$V(G)_{Jode}$	0.03
D_{Jode}	0.05
E_{Jode}	0.1
MI_{Jode}	0.20

(c)

	G_{AllDec}
LOC_{AllDec}	0.21
$V(G)_{AllDec}$	0.19
D_{AllDec}	0.13
E_{AllDec}	0.16
MI_{AllDec}	0.03

(d)

Figure 8.3: Spearman rank coefficient correlation ρ between the classical metrics and the resilience of obfuscated programs using all decompilers (*JD, JAD*, and *Jode*).

Overall, the Spearman rank correlation of the classical measures versus G_{AllDec} are positive, and all the p-values are below 0.05, which means the correlations are significant. However, observing the correlation per individual decompiler, we notice a very small correlation in G_{Jode} of 0.04. We checked for significance in Fig. 8.4, and we found that the p-value are above 0.05 threshold, apart from the Halstead effort metric (E), which scores exactly 0.05. We conclude, based on this analysis that the classical complexity metrics, $LOC, V(G), D, E$ and *MI* do not correlate with the percentage of code obfuscation resilience based on *Jode* decompiler, G_{Jode}. We investigate this matter in more detail, and find the main reason for this decrease in complexity: *Jode* fails to produce a complete decompilation when it decompiles obfuscated programs of SPECjvm2008 with arbitrary byte-code, such as *BuggyCode (SM-C-BC)* (see Table 7.4).

	G_{JAD}
LOC_{JAD}	$\ll 0.05$
$V(G)_{JAD}$	$\ll 0.05$
D_{JAD}	$\ll 0.05$
E_{JAD}	$\ll 0.05$
MI_{JAD}	$\ll 0.05$

(a)

	G_{JD}
LOC_{JD}	$\ll 0.05$
$V(G)_{JD}$	$\ll 0.05$
D_{JD}	0.01
E_{JD}	0.012
MI_{JD}	$\ll 0.05$

(b)

	G_{Jode}
LOC_{Jode}	0.46
$V(G)_{Jode}$	0.54
D_{Jode}	0.33
E_{Jode}	0.05
MI_{Jode}	0.54

(c)

	G_{AllDec}
LOC_{AllDec}	$\ll 0.05$
$V(G)_{AllDec}$	$\ll 0.05$
D_{AllDec}	$\ll 0.05$
E_{AllDec}	$\ll 0.05$
MI_{AllDec}	$\ll 0.05$

(d)

Figure 8.4: The p-value of each Spearman rank coefficient correlation between the classical metrics and the resilience of obfuscated programs using all decompilers (*JD, JAD*, and *Jode*).

[3] Although, *NC* is a normalised measure of *C*, we are also interested in comparing compression length measure (*C*), as an absolute measure, with other classical complexity metrics.

	G_{JAD}
$Pt_{LOC_{JAD}}$	0.20
$Pt_{V(G)_{JAD}}$	0.20
$Pt_{D_{JAD}}$	0.23
$Pt_{E_{JAD}}$	0.26
$Pt_{MI_{JAD}}$	0.12

(a)

	G_{JD}
$Pt_{LOC_{JD}}$	0.19
$Pt_{V(G)_{JD}}$	0.19
$Pt_{D_{JD}}$	0.15
$Pt_{E_{JD}}$	0.14
$Pt_{MI_{JD}}$	0.16

(b)

	G_{Jode}
$Pt_{LOC_{Jode}}$	-0.02
$Pt_{V(G)_{Jode}}$	-0.06
$Pt_{D_{JAD}}$	-0.03
$Pt_{E_{JAD}}$	0.1
$Pt_{MI_{JAD}}$	-0.18

(c)

	G_{AllDec}
$Pt_{LOC_{AllDec}}$	0.15
$Pt_{V(G)_{AllDec}}$	0.12
$Pt_{D_{AllDec}}$	0.13
$Pt_{E_{AllDec}}$	0.18
$Pt_{MI_{AllDec}}$	0.07

(d)

Figure 8.5: Spearman rank coefficient correlation ρ of each classical metric potency (*Pt*) and the resilience of obfuscated programs using all decompilers (*JD, JAD*, and *Jode*).

In the potency measure, using classical complexity metrics (Section 7.3.3), the overall Spearman rank correlation obtains a positive value with p-values below the 0.05 threshold (significant). The Spearman rank correlations, in case of *Jode*, produce negative values; however, they are not significant (see Fig. 8.6) as their p-values are above 0.05, with only one exception in Pt_{MI} versus G_{Jode} (see Fig. 8.6).

	G_{JAD}
$Pt_{LOC_{JAD}}$	$\ll 0.05$
$Pt_{V(G)_{JAD}}$	$\ll 0.05$
$Pt_{D_{JAD}}$	$\ll 0.05$
$Pt_{E_{JAD}}$	$\ll 0.05$
$Pt_{MI_{JAD}}$	0.02

(a)

	G_{JD}
$Pt_{LOC_{JD}}$	$\ll 0.05$
$Pt_{V(G)_{JD}}$	$\ll 0.05$
$Pt_{D_{JD}}$	$\ll 0.05$
$Pt_{E_{JD}}$	0.01
$Pt_{MI_{JD}}$	0.02

(b)

	G_{Jode}
$Pt_{LOC_{Jode}}$	0.68
$Pt_{V(G)_{Jode}}$	0.28
$Pt_{D_{JAD}}$	0.64
$Pt_{E_{JAD}}$	0.1
$Pt_{MI_{JAD}}$	0.02

(c)

	G_{AllDec}
$Pt_{LOC_{AllDec}}$	$\ll 0.05$
$Pt_{V(G)_{AllDec}}$	$\ll 0.05$
$Pt_{D_{AllDec}}$	$\ll 0.05$
$Pt_{E_{AllDec}}$	$\ll 0.05$
$Pt_{MI_{AllDec}}$	0.02

(d)

Figure 8.6: The p-value of Spearman's rank coefficient correlation between of each classical metric potency (*Pt*) and the resilience of obfuscated programs using all decompilers (*JD, JAD*, and *Jode*).

We answer $\mathbf{H}_{02}$ by comparing Spearman rank correlations between the proposed metrics and the classical complexity measures. We can observe that the p-values of Spearman rank correlation in the sets of measurements, after decompiling with all decompilers: (*NCD*, π_U, *NC* , *C*), (*LOC*, *V(G)*, *D*, *E* and *MI*) and (Pt_{LOC}, $Pt_{V(G)}$, Pt_D,Pt_E and Pt_{MI}) against G_{AllDec} are below 0.05. We then move on and use the correlations to perform the comparison; we see that all the proposed metrics, apart from *NC*, outperform the classical complexity metrics and their potency measures. Therefore, we reject the null hypothesis $\mathbf{H}_{02}$ and accept the alternative hypothesis $\mathbf{H}_{12}$, with only one exception for *NC*, as it scores less than all of the classical complexity metrics.

8.2.4 Generic Linear Regression Model for Code Obfuscation Security

So far, we present the empirical evaluation of the proposed metrics. The analysis of the results in Section 8.2.2 shows the usefulness of these metrics to reflect the degree of resilience of obfuscation

techniques. We now construct a multi-linear regression model to produce a single quantitative value from all of the proposed metrics, and investigate how well the data fits this model. The multi-linear regression equation has the following form:

$$Y = b + a_1X_1 + a_2X_2 + a_2X_3$$

a_1, a_2 and a_3 are the coefficient or the slopes, b normally referred to as the *intercept* or *constant*,[4] X_1, X_2 and X_3 are the independent variables (our proposed metrics) and Y is the dependent variable that we are aiming to predict using the independent variables.

In Section 6.4.6 we proposed that model, discussed also in Section 7.5, for estimating the security of code obfuscation using a multi-linear regression model. This model is useful to express the quality of obfuscated code using a single quantitative value S. We used our proposed metrics as independent variables to estimate S. This model is only designed for Java programs (the subject of the study) attacked by decompilation.

We established the best-fitting multi-linear regression model in Fig. 8.7 by computing a regression equation, providing a mathematical relation between NCD, π_U, NC and G_{AllDec}. The outcome of estimated multi-linear regression model is given by the following equation based on the regression analysis:

$$S = 73.26 + 41.32 \cdot NCD + 0.77 \cdot \pi_U - 26 \cdot NC$$

	Coefficient	Standard Error	p-Value	R Square	Adjusted R Square
Intercept	73.26	1.53	$\ll 0.05$		
NCD	41.32	0.8	$\ll 0.05$		
π_U	0.77	0.54	0.152	0.71	0.71
NC	-26	2.24	$\ll 0.05$		

Figure 8.7: Regression analysis for the combined proposed metrics with versus all G_{AllDec}.

In Fig. 8.7 we report the regression analysis of the proposed metrics; the coefficient field shows all the slope values that are used to construct the regression equation. The standard error field assesses the precision of the predictions, i.e. the confidence interval of predication. The p-value field reports if the coefficients are significant. R^2(*R-Square*) measures how close the actual data are to the fitted regression line (estimated data). Intuitively, it can be explained as the percentage

[4]It represents the value of a regression equation if all other variable are zeros.

of variation in the dependent variable, G_{AllDec}, with respect to the data that is accounted for by the regression model (the proposed metrics). Formally, given a data sample of n observed values, $y_1, ..., y_n$, each associated with a predicated (fitted) value $\hat{y}_1, ..., \hat{y}_n$, where $\tilde{y}$ is the mean (average) [KKM88].

$$R^2 = \frac{\sum_{i=1}^{n}(\hat{y}_i - \tilde{y})}{\sum_{i=1}^{n}(y_i - \tilde{y})}$$

The adjusted version of R^2 (R^2_{adj}) adjusts the statistic based on the number of independent variables in the model that significantly affect the dependent variable. It is defined in terms of R^2 as follows, where n is the number of observations in the data set, and p is the number of independent variables.

$$R^2_{adj} = 1 - (1 - R^2) \cdot \frac{n-1}{n-p-1}$$

Having the regression equation, we can predict the dependent variable (G_{AllDec}) using the independent variables (NCD, π_U, NC); however, we need a mechanism to test how the data fits this model. Generally, the regression model fits the data properly if the differences between the observed values and the model's predicted values are minimal.

The outcome of R^2 and its adjusted version shows that the regression model is able to estimate over 71% of the variation in the dependent variable G_{AllDec} (observed data); it is considered as adequate estimate, because we do not expect the model to predicate 100% of variations in dependent variable.

The estimated regression model is fitted using a *cross validation* technique to avoid *overfitting* [SL12] in the predicted data.[5] The goal of a cross-validation is to measure the predictive performance (ability to predict) of the regression model as a high R^2 value may not necessarily means the model is good. We use the *K-Fold* [SL12] cross-validation technique to perform the validity checking, which randomly breaks the dataset into *K* partitions and performs the analysis, by calculating the sum of errors using *Mean Square Error (MSE)* and *Root Mean Square Error (RSME)* i.e. $RMSE = \sqrt{MSE}$. In the context of regression analysis, the mean square error measures the average squares of error variances or deviations of the predictor estimation from the correct

[5] Overfitting generally occurs when a model is excessively complex relative to the amount of data available. An overfitted model performs much worse (more errors) on the test dataset than on the training dataset, resulting in poor predictive performance.

values, i.e. it measures how close a fitted line (the regression model) is to the data points. The squared is done so that the negative values do not cancel the positive values. Given a data sample of n observed values, $y_1, ..., y_n$, and their (predicated) fitted values $\hat{y}_1, ..., \hat{y}_n$.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

We conduct the cross validation using $K = 10$ folds [6]. The outcome of the analysis is presented in Fig. 8.8; it shows that *RMSE* scores on average around 10.39 ± 1.12 for all the tested folds. This can be interpreted as the model miss-predicts the dependent variable (the code obfuscation resilience to decompilers) by 10.39% (the dependent variable is a percentage too).

K	*MSE*	*RMSE*
1	86.12	9.28
2	117.56	10.84
3	132.18	11.49
4	122.61	11.07
5	72.48	8.51
6	118.26	10.87
7	83.62	9.14
8	141.7	11.9
9	96.56	9.82
10	120.65	10.98

Figure 8.8: Cross validation results on the adjusted regression model.

Investigating the standard error of regression shows 1.53% of errors occur for the intercept (the constant), 0.8% of errors on average for the estimated coefficient for *NCD*, and 0.54% for π_U. In *NC*, we observe a relatively high score of errors for the coefficients: 2.24% comparing to *NCD* and π_U.

We check the p-value for each of the estimated coefficients, which are computed using t-statistic [She07]. It tests the null hypothesis that the regression coefficients are not equal to zero. The results in Fig. 8.7 shows that the p-value of the coefficients for *NCD* and *NC* are satisfactory, as they score below the 0.05 threshold of the significant level; apart from π_U which achieves 0.15 above the threshold, providing evidence that π_U does not contribute significantly to the regression model.

We adjust the above equation to find the predictors (independent variables) that contribute significantly to the model. We reduce independent variables into two by eliminating π_U using *Stepwise regression analysis* [She07]; this type of statistical analysis is conducted by sequentially adding predictors to the model, based on their significance until a satisfactory model is found. The

[6] $K = 10$ folds is commonly used in the most of cases in linear regression model cross-validation.

results of the stepwise regression analysis are in Fig. 8.9, the modified regression equation is given below

$$S = 73 + 41.56 \cdot NCD - 26 \cdot NC$$

	Coefficient	Standard Error	p-Value	R Square	Adjusted R Square
Intercept	73	1.52	$\ll 0.05$		
NCD	41.56	0.78	$\ll 0.05$	0.71	0.71
NC	-26	2.23	$\ll 0.05$		

Figure 8.9: Stepwise regression analysis results for adjusting the regression model of proposed metrics with versus all G_{AllDec}.

We notice that eliminating π_U does not reduce the values of R^2, nor the standard error, for all coefficients. Furthermore, we present diagnostic plots of the residuals, the value of errors miss-predicting the correct data i.e. Residual = Observed value - Predicted value, to confirm that the model is indeed satisfactory, (see Fig. 8.10).

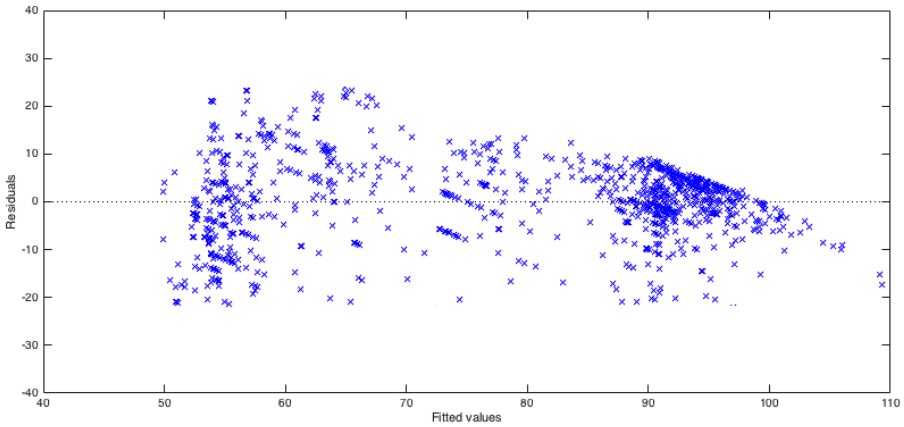

Figure 8.10: Plot of residual versus the fitted values

Analysing the residuals helps to confirm that the model is also satisfactory. If the residuals appear to behave randomly, it suggests that the model fits the data well. The residuals should approximately fall into a symmetrical pattern and have a constant spread throughout the range [She07]. We find that around 90% of residuals have constant variance at around the mean 0 using a scatter-plot of residuals in Fig. 8.10. Despite that, we notice some bias in the residuals for a part of fitted values. This suggests that we may expect some advantages using non-linear regression models [KKM88], which may help to improve the accuracy of data predication; although given the distribution of the residual (see Fig. 8.10), the improvement is possibly limited. This requires

further study which we reserved for future work.

8.3 Measuring the Quality of code obfuscation

In the previous sections we have provided statistical evidence to evaluate (*NCD*, π_U, and *NC*) as valid metrics to measure the quality of code obfuscation, and proposed a multi-linear regression model to provide a quantitative single value for the quality of code obfuscation. In this section, we apply these metrics to answer the following set of the research questions. These questions are concerned with measuring the quality of code obfuscation, per individual obfuscation and according to their types (Section 7.3.6), in addition to their resilience to decompilation attacks.

(**RQ**$_3$) Is there any change in the Kolmogorov (compression) complexity measures between a clear code and its obfuscated version using different obfuscation algorithms? Does that change, if it occurs, implies an increase in complexity measure, and in which magnitude?

(**RQ**$_4$) Does the obfuscation process produce any changes (measured using *NCD*) in the clear unobfuscated code, and in which magnitude?

(**RQ**$_5$) What is the effectiveness of obfuscation algorithms using the proposed measure, by type: Control-Flow, Data and Layout obfuscation?

(**RQ**$_6$) What is the impact of deobfuscation (decompilers) on code obfuscation resilience?

8.3.1 Hypothesis Formulation

The above set of the research questions deals with using the proposed metrics to study the quality of code obfuscation. Among these questions, we formulate the null hypotheses for the questions **RQ3** and **RQ4**, the rest of the questions are answered based on the descriptive statistic (average) and visual data inspections using graphs. For question **RQ3**, two null hypotheses are set as follows:

- Null hypothesis, $\mathbf{H}_{03a}$: There is *no difference* in the Kolmogorov complexity (compression) between clear and obfuscated code i.e. for an obfuscated program $\mathcal{O}(P,\lambda)$ of a program P, $C(\mathcal{O}(P,\lambda)) = C(P)$.

  Alternative hypothesis: $\mathbf{H}_{13a}$: There is a *difference* in the Kolmogorov complexity (compression) between clear and obfuscated code i.e. for an obfuscated program $\mathcal{O}(P,\lambda)$ of a

program P, $C(\mathcal{O}(P,\lambda)) \neq C(P)$.

Upon testing this hypothesis, we proceed and check whether the difference, if it exists, is a result of a positive increase. Therefore, we formulate the following null hypotheses:

- Null hypothesis, $\mathbf{H}_{03b}$: There is *no increase* in the Kolmogorov complexity (compression) of clear code due to obfuscation.

  Alternative hypothesis: $\mathbf{H}_{13b}$: There is an *increase* in the Kolmogorov complexity (compression) of clear code due to obfuscation.

To answer question **RQ4**, we use the following null hypothesis:

- Null hypothesis, $\mathbf{H}_{04}$: There are *no changes* in the clear code due to obfuscation i.e. for an obfuscated program $\mathcal{O}(P,\lambda)$ of a program P, $NCD(P,\mathcal{O}(P,\lambda)) = 0$.

  Alternative hypothesis, $\mathbf{H}_{14}$: There are *changes* in the clear code due to obfuscation i.e. for an obfuscated program $\mathcal{O}(P,\lambda)$ of a program P, $NCD(P,\mathcal{O}(P,\lambda)) \neq 0$.

8.3.2 Obfuscation Analysis using *NCD*

We use the normalised compression distance (*NCD*) to study the quality of obfuscation. The reported data of this measure is presented in Fig. 8.11. We start by answering the hypothesis $\mathbf{H}_{04}$ of question **RQ4**. It concerns with checking whether obfuscation produces changes to the clear code. The reason for including this question is to check whether the clear programs (the benchmark) are sensitive to obfuscation using different obfuscation algorithms, without applying our attack (decompilers). Although we can visually answer this question by inspecting Fig. 8.11, we apply the Mann-Whitney statistical test (see Section 7.5) to check whether we can reject the null hypothesis (when the p-value is less than 0.05). The outcome of the Mann-Whitney test shows that we can reject the null hypothesis and accept the alternative hypothesis, as the p-value is smaller than 0.05. This shows that *NCD* is a non-zero value for all obfuscated programs. Furthermore, we test the *NCD*, using the Mann-Whitney test as well, for all the decompiled obfuscated code using*JAD, JD* and *Jode* i.e. *NCD* between the clear decompiled code and the decompiled obfuscated code. The results show that *NCD* is also positive.

It is clear that obfuscation creates changes to clear code; we investigate the obfuscation effectiveness before decompilation using *NCD* in Fig. 8.11. First, it is noticed that the *NCD* (lines

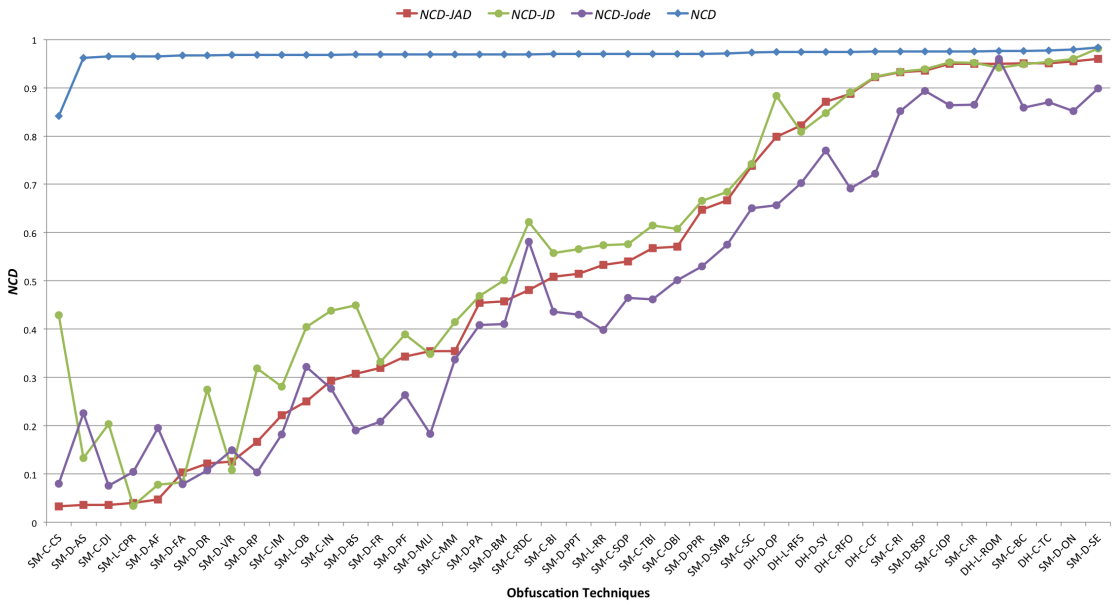

Figure 8.11: Averaged *NCD* for all obfuscation techniques with (*NCD*) and without decompilation attack (*NCD-JD, NCD-JD,* and *NCD-Jode*)

labeled with *NCD*) between the clear code and the obfuscated code, using all the obfuscated techniques, have similar values. However, these values are close to 1 (*NCD* = 0.98), which means that obfuscation has made substantial changes in the clear code of SPECjvm2008 benchmark, making the obfuscated code totally different from the original clear code. Only one technique shows a different behaviour: *SM-C-CS*, a class splitter obfuscation technique, a part of Sandmark's obfuscator techniques.

Analysis Per Obfuscation Type

We analyse the obfuscation transformation algorithms according to the type of transformations (see Section 7.3.6): Control-Flow, Data and Layout, and report the average *NCD* for each type after we subject the obfuscated programs of SPECjvm2008 to decompilation. The aim is to answer the part of question **RQ4** that is related to *NCD* measure. We see the results in Fig. 8.12 : Layout obfuscation techniques perform better than Control-Flow and Data obfuscation. The Layout obfuscation algorithms, specially Dasho's techniques, produce more changes in the benchmark programs than the Control-Flow and Data obfuscation types. We investigate this matter in more detail; it is noticed that Dasho's Layout obfuscation techniques (based on Overload-Induction) apply intensive renaming for identifiers and variables, which render the decompilation process in

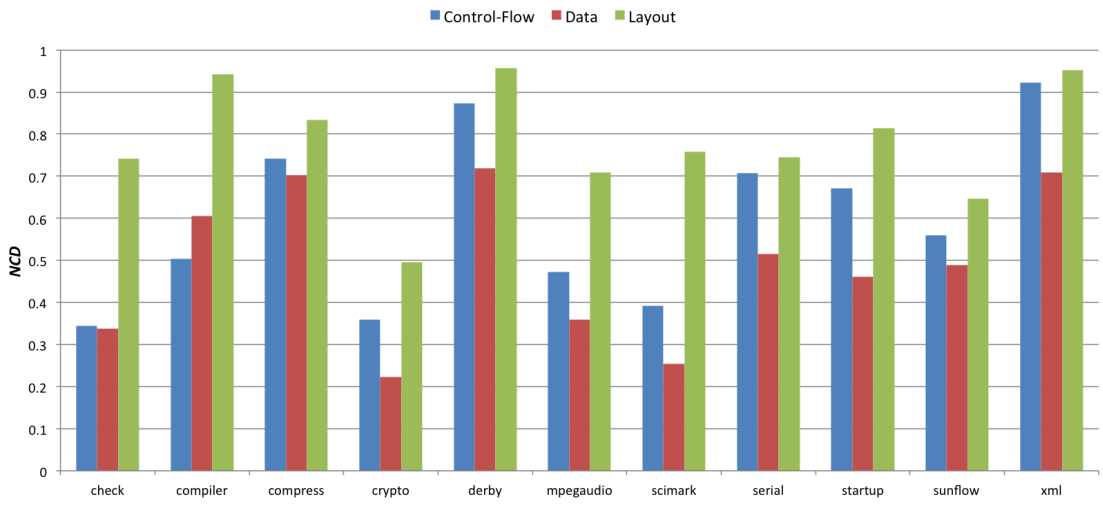

Figure 8.12: Averaged *NCD* measure per decompiled obfuscation transformation. Control-Flow, Data and Layout obfuscation.

many instances useless. Therefore, this type of obfuscation algorithms overload the result of*NCD* between the clear code and the deobfuscated programs using decompilation.

The Impact of Decompilation on Obfuscation Resilience

We answer question **RQ5** by investigating the effect of decompilation as an attack on code obfuscation resilience using *NCD*. Fig. 8.11 reports the results: it shows three different lines labeled with *JAD, JD* and *Jode* which resemble the average *NCD* of obfuscation techniques after being subjected to decompilation.

Overall, we observe a significant decrease in the *NCD* of all obfuscated programs due to decompilation. This decline in *NCD* varies among the different obfuscation techniques, which also indicates different behaviour in terms of resilience to decompilation process. It is difficult, based on Fig. 8.13, to clearly distinguish the resilience of obfuscations among the different decompilers, as they score almost similar results. However, most of the obfuscation techniques show the highest resilience against *JD* with around 50% more than *JAD*, and around 80% more than *Jode*. We further investigate the resilience of each obfuscation techniques against all decompilers: *String Encoder (SM-D-SE)* shows the highest resilience for *JD* and *JAD*, where *Rename OverInduction Maintainhierarchy (DH-L-ROM)* shows the highest resilience for *Jode*. *ClassSplitter (SM-C-CS)* demonstrates the lowest resilience for *JAD*, and *Constant Pool Reordering (SM-L-CPR)* shows the weakest against *JD*. In the case of *Jode*, *Field Assignment (SM-D-FA)* scores the weakest resilience.

8.3.3 Obfuscation Analysis using π_U

In this section we report on the analysis of obfuscation unintelligibility index (π_U), the algorithmic complexity difference between clear and obfuscated code (see Definition 6.22). We answer the question **RQ3** in Section 8.3 by investigating its formulated null hypothesis $\mathbf{H}_{03a}$, which states that there is no difference in the Kolmogorov complexity (approximated by compression) between clear and obfuscated code; we test if $\pi_U = 0$ which is equivalent to $C(\mathcal{O}(P, \lambda)) = C(P)$, see Section 6.4.1. Visually, in Fig. 8.13, we observe that all obfuscation techniques produce changes in the complexity of obfuscated code, as the upper line labeled with *Un-Index* does not produce any zero value of π_U. The visual results in Fig. 8.13 are reported on average, by computing the mean (see Section 7.5). However, by applying the Mann-Whitney statistical test we can get even better statistical evidence (multiple pairwise comparisons) that supports the visual investigation, and subsequently checks whether we can reject the null hypothesis. The outcome of the Mann-Whitney test shows that we can reject the null hypothesis with high statistical significance, as the p-value is below 0.05.

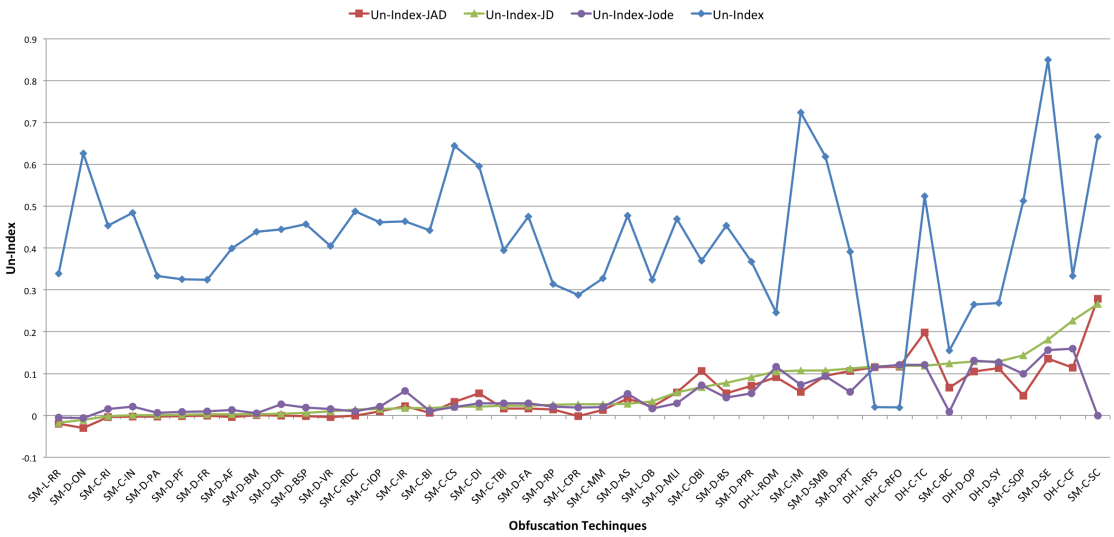

Figure 8.13: Averaged Un-Index for all obfuscation techniques with and without decompilation attack.

In the case of checking the changes in the kolmogorov complexity after the decompilation, we could not infer this result visually as we see in Fig. 8.13. Using the Mann-Whitney statistical test, we reject the null hypothesis that the complexity of the deobfuscated programs (decompiled) are not different than their clear unobfuscated code.

While the statistical test of $\mathbf{H}_{03a}$ allows for checking the presence of significant differences, it does not provide any information about the magnitude of such a difference. We proceed to investigate to what extent the use of obfuscation changes the complexity of source-code i.e. the magnitude of changes in the complexity (π_U), and whether they are positive or not, by checking the null hypothesis $\mathbf{H}_{03b}$. The outcome of the Mann-Whitney statistical test shows that the magnitude of changes in complexity, due to the obfuscation process, is always positive with high statistical significance, as their p-value report below 0.05. Similarly, we notice the same results that show the positive magnitude (on average) for most of the deobfuscated programs ($\pi_U > 0$).

Analysis Per Obfuscation Type

This section answer the part of question **RQ4** which is related to the π_U measure (see Section 8.3). The results are presented in Fig. 8.14. Control-Flow obfuscation techniques outperform both Data and Layout obfuscation techniques in most of the benchmark programs (9 out of 11). That was a bit surprising, as we expected the Data obfuscation to exceed both Control-Flow and Layout obfuscation in terms of complexity. This is due to the nature of Data obfuscation that adds a lot of noise to a program's data structure comparing to Control-Flow obfuscation type, which only complicates the structure of Control Flow Graph (CFG). We check the main reason behind this behaviour; we notice that most of these algorithms in Dasho and Sandmark add a lot of spurious branches full of random code, especially opaque predicates (see Section 3.4.2). Layout obfuscations score the lowest (8 out of 11 benchmark programs) as the most of these algorithms rely on renaming techniques, especially in Dasho's obfuscation techniques; Dasho obfuscator renames as many methods as possible to exactly the same name. Apparently, this produces a high level of redundancy and regularity, which reduces the complexity of the code. Furthermore, in particular, Dasho's obfuscation techniques employ, in addition to renaming, heavy optimisation methods.

The Impact of Decompilation on Obfuscation Resilience

In this section we proceed to answer question **RQ5** in Section 8.3 using the π_U as a measure for studying the effect of decompilation attacks on the obfuscated benchmark programs. In Fig. 8.13, we can see that decompilation successfully managed to decrease the π_U of the obfuscated benchmark programs, for the majority of obfuscation techniques. However, we notice in two Dasho's obfuscation techniques: *Rename Flattenhierarchy Simple (DH-L-RFS)* and *Rename*

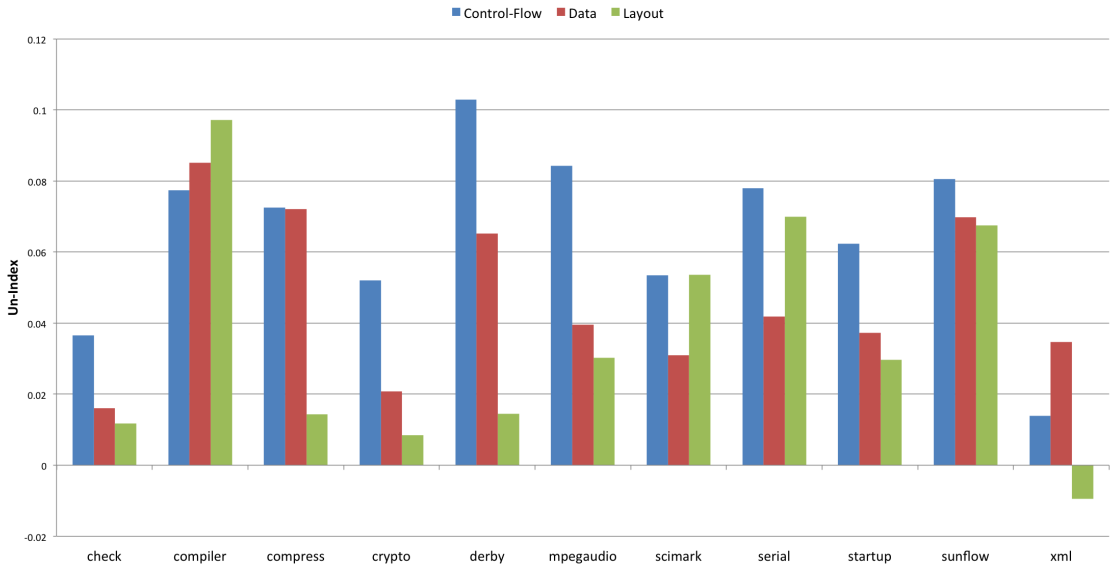

Figure 8.14: Averaged Un-Index (π_U) measure per decompiled obfuscation transformation. Control Flow, Data and Layout obfuscation.

FlattenHierarchyOverInduction (DH-C-RFO) their decompilation increases the π_U above their original π_U. To explain this behaviour, we see on the one hand that both of those two obfuscation techniques employ heavy use of renaming; most of the renaming replaces variables and identifiers with less complex ones, normally adding the same and simple unreadable identifiers. On the other hand, decompilers replace the obfuscated variables and identifiers with more complex names.[7]

Rename Register (SM-L-RR) and *Overload Names (SM-D-ON)* achieve the lowest π_U of all deobfuscated benchmark programs, as the decompilers replace all overloaded methods and local registers (byte-code level) into less complex and randomless names. *String Encoder (SM-D-SE)* shows the highest scores of π_U before decompilation. The reason behind this is that *String Encode* uses an encryption process which produces a lot of randomness in the code. However, decompilers are effective at reducing the π_U of *SM-D-SE*, but not to the level of other deobfuscated (decompiled) programs (different obfuscation techniques). We investigate this reduction in the π_U, and noticed that the reason was not due to the effectiveness of decompilers producing correct and less complex program, but was related to the fact that decompilers failed to produce the correct code out of obfuscated one.

[7]The names replaced by the decompilers (*JD, JAD* and *Jode*) are more complex but more readable than the obfuscated variable names.

8.3.4 Obfuscation Analysis using *NC*

We use the normalised compression (*NC*) to study the impact of obfuscation on clear code. The reported data of this measure is presented in Fig. 8.16. Around 50% of obfuscation techniques, of different obfuscation type, score similar normalised compression values (*NC=0.91*) with very minor differences, and all of these techniques are part of the Sandmark obfuscator framework. This could indicate a common design pattern among these techniques, which needs further investigation. Dasho's obfuscation techniques show different behaviour: only Data obfuscation type techniques have similar *NC* values. We also find that *String Encoder (SM-D-SE)* and *BuggyCode (SM-C-BC)* perform better than all the obfuscation techniques in terms of *NC*, where *Class Splitter (SM-C-CS)* and *Rename FlattenHierarchy OverInduction (DH-C-RFO)* score the lowest among all obfuscation techniques.

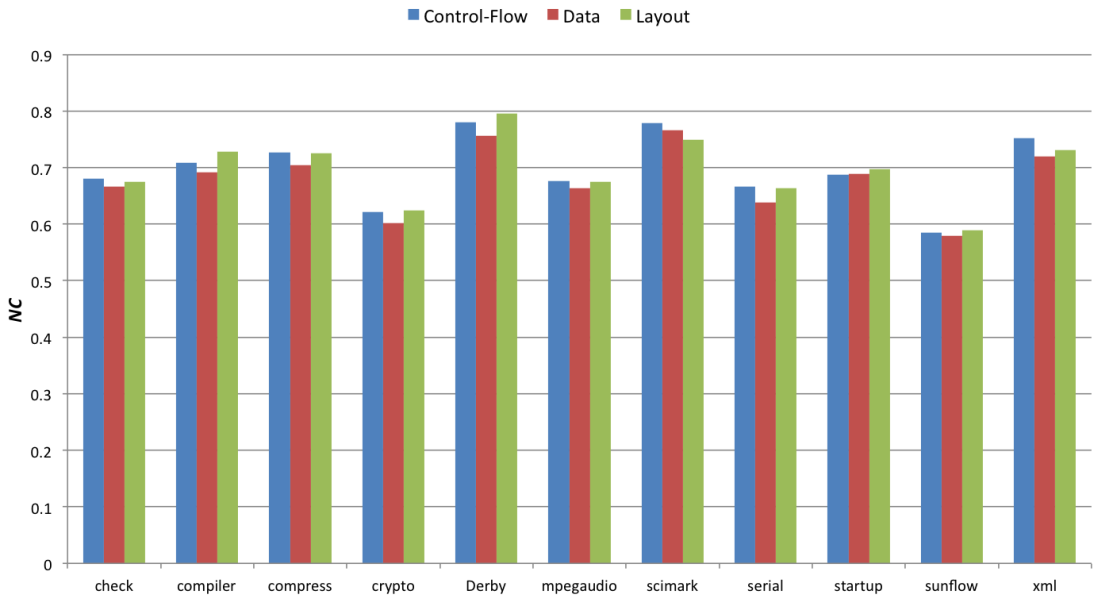

Figure 8.15: Averaged *NC* measure per decompiled obfuscation transformation. Control-Flow, Data and Layout obfuscation.

Aggregating the obfuscation techniques according to their types shows a very minor difference using the proposed metric (see Fig. 8.15). Layout and Control-Flow obfuscation performed roughly the same, slightly exceeding Data obfuscation techniques.

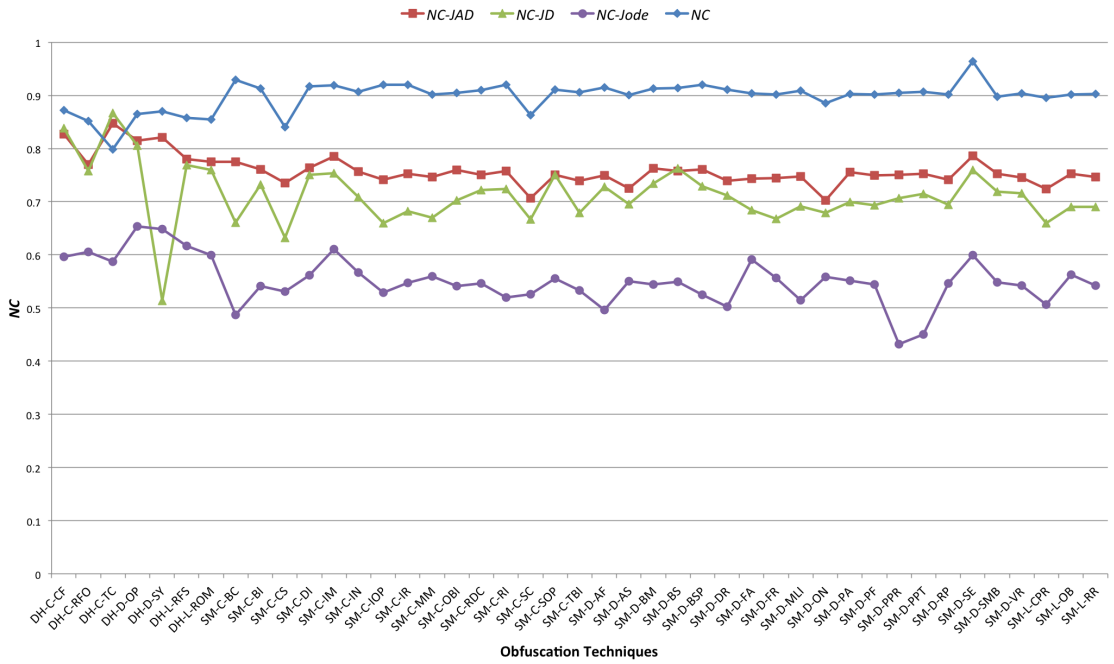

Figure 8.16: Averaged *NC* for all obfuscation techniques with and without decompilation attack (*NC-JD*,*NC-JD,* and *NC-Jode*).

The Impact of Decompilation on Obfuscation Resilience

We study the effect of decompilation as an attack on code obfuscation resilience using *NC* as a measure. Fig. 8.16 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, show high resilience against *JD* and *JAD*, and 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 (DH-D-SY)* is a technique designed to fail decompilation; however *JD* was very effective at thwarting this technique. *StringEncoder (SM-D-SE)* has the highest resilience against *JD* where *Class Splitter (SM-C-CS)* 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. 8.16. *Jode* is very effective at reducing the complexity of *TryCatch10 (DH-C-TC)* obfuscated programs comparing to *JAD* and *JD*. We notice that *Jode* managed to reduce the *NC* to the benchmark baseline level in Fig. 8.16. We check this

matter in more details, and find the main reason for this decrease in normalised complexity: *Jode* failed to produce a complete decompilation when it decompiles the programs that were obfuscated with arbitrary byte-code, such as *BuggyCode (SM-C-BC)*. We also realised the same problem with *JAD*; surprisingly *Jode* failed to replace `java.lang.Integer` object to the correct *int* in the source code for *Promote Primitive Register (SM-D-PPR)*, and *Promote Primitive Type (SM-D-PPT)* obfuscation, which agrees with Hamilton et al.[HD09] observation that *Jode* sometimes fails at resolving and inferring the correct types. Nevertheless, *Jode* decompiles the other obfuscated programs with a reasonable accuracy. In general, all decompilation managed to reduce *NC* to a certain degree, where *Jode* outperforms all the decompilers at reducing the complexity of obfuscated programs.

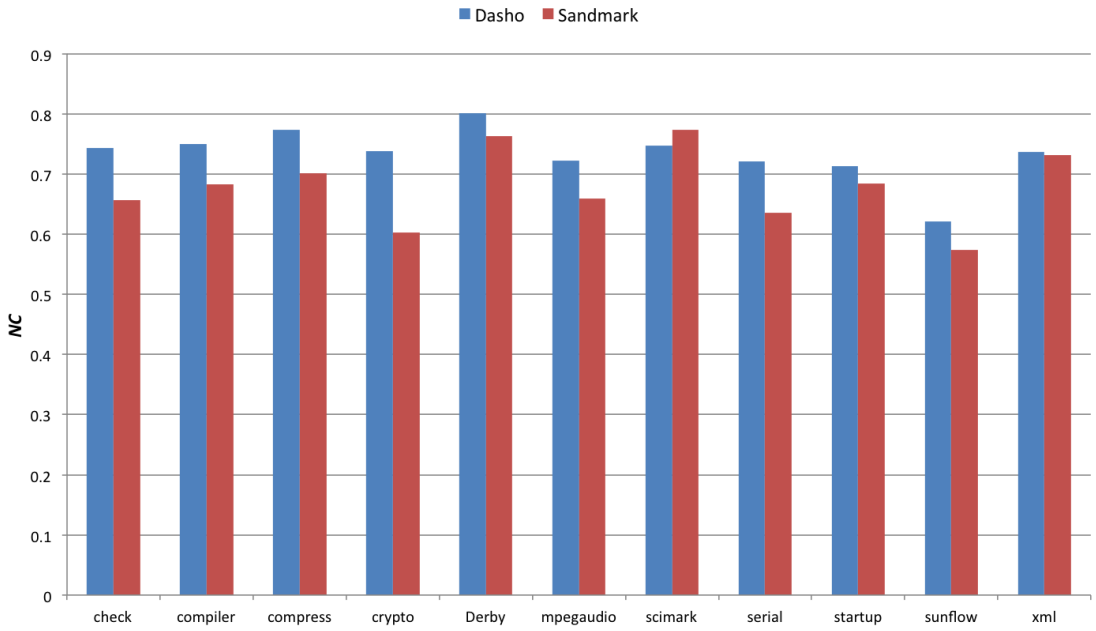

Figure 8.17: Averaged *NC* measure for all decompiled obfuscation techniques according to obfuscators.

8.3.5 Discussion on the Findings

In this section we elaborate more on the outcome of the experiment, and discuss the implications of the finding as a result of answering the posed research questions. The *NCD* shows that all the obfuscation algorithms produce drastic changes in the original code; the amount of these changes are similar among most of the obfuscated programs. However, the *NCD* measure behaves totally different when the obfuscated code is subjected to an attack. The deobfuscation (using decompliers)

responds distinctively to each individual obfuscated technique; *NCD* reflects this behavior by reporting different values. This means that each obfuscation technique provides different resilience to the attacks.

We can notice the same behaviour using unintelligibility index π_U. Increasing Kolmogorov complexity, which can be monitored using π_U, shows the different resilience of obfuscated code to the decompilers. Higher π_U before decompilation did not match the increase in the π_U after. However, in case of normalised compression *NC*, after decompilation, the deobfuscated programs have the same trends as the obfuscated programs, but *NC* correlates poorly with the obfuscation resilience using G_{AllDec}, where simple compression length (C), along other classical complexity metrics correlate better with G_{AllDec}.

In general, this shows the danger of relying on absolute metrics as a means to predict the obfuscation resilience, without taking into account the attacker itself when we measure. For example, if the goal of the defender is to obfuscate a program so that it becomes difficult to decompile, then the defender has to test the obfuscated program against a decompiler. Subsequently, the outcome of the decompiler, which is code as well, can be measured using the proposed metrics to estimate the obfuscation resilience.

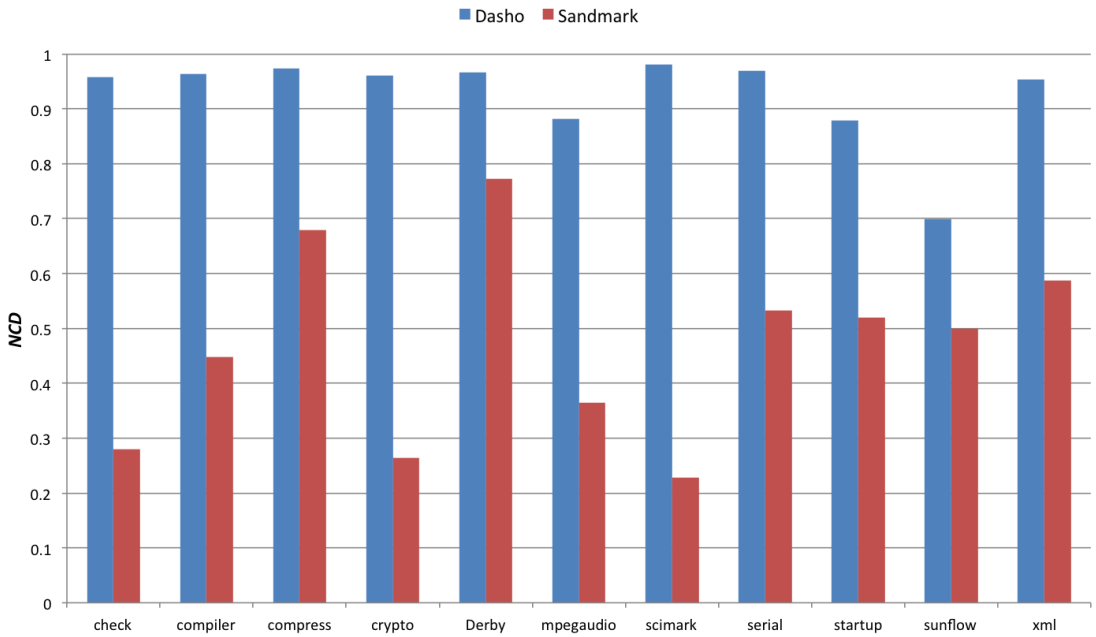

Figure 8.18: Averaged *NCD* measure for all decompiled obfuscation techniques according to obfuscators.

We find that the initial complexity of clear code affects π_U, especially when we investigate **RQ3**. In some of the benchmark programs, the compression length is relatively big (high Kolmogorov complexity) so that the obfuscator has a little room to increase the complexity. We see that in Fig. 8.14 for benchmark programs: crypto, compress, and xml. For example, if the clear program contains a small number of string variables, then the obfuscation using *String Encoder (SM-D-SE)* becomes insignificant. Similarly, if the code does not contain generics or arrays, then the obfuscation techniques that manipulate these data structures become ineffective which can undermine the potential of code obfuscation having effective protection. Therefore, the defender has to decide whether the clear code needs to be obfuscated or not, and which feature to obfuscate, probably by using Kolmogorov complexity (compression length) as a 'sanity check' metric to test the initial complexity.

Finally, our finding confirms the theoretical reasoning in Section 4.2.4: increasing the complexity of obfuscated does not always produce increases in the *NCD*, this can be seen in Fig. 8.11 and Fig. 8.13. In many instances, such as *Rename Flattenhierarchy Simple (DH-L-RFS)* and *Rename FlattenHierarchy OverInduction (DH-C-RFO)* which obtain the lowest Kolmogorov complexity (compression), their *NCD* measures are relatively high; this case is typical in Dasho's obfuscation techniques. In general, we see from Fig. 8.18, 8.17, and 8.19 that Dasho's obfuscation techniques outperformed Sandmark's obfuscation techniques using all proposed measures.

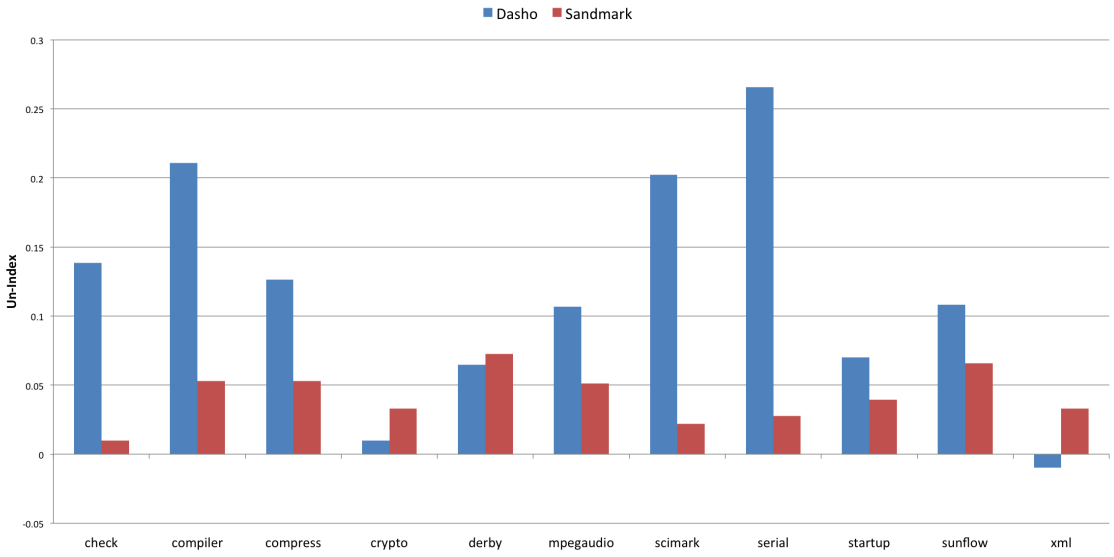

Figure 8.19: Averaged Un-Index (π_U) measure for all decompiled obfuscation techniques according to obfuscators.

8.4 Limitations and Threats of Validity

In this part we address the potential threats to the experiment that may affect the outcome of our results. We investigate these possible issues using Wohlin et al.[WRH+00] general framework for threats validation in software engineering experiments. The framework consists of four major classes: internal, construction, conclusion and external validity threats.

8.4.1 Internal Validity

Internal validity demonstrates the extent in which our operations are sound and accurate. It checks for the effect of additional factors that may or may not account for which negatively influence our results.

Compression acts as an upper bound estimation for Kolmogorov complexity; however, it is safe for large binary sequences (of length $l \geq 50$ bits) [STZDG12]. For short binary sequences, compression becomes inefficient; many researchers are trying to tackle this limitation by providing alternative solutions based on the *Coding theorem* such as [GZD11]; the Coding theorem connects algorithmic probability (or frequency of production) [LV08] to Kolmogorov complexity for computing effective complexity estimation for short binary strings. This approach is beyond the scope of this thesis; in our case, all the benchmarks programs are large in size.

We also employ plagiarism detection technique (Sherlock) to measure decompilation similarity between the original decompiled benchmark programs and its obfuscated versions. Sherlock is based on syntactic matching; it is used, in our experiment, to measure the effectiveness of decompiled obfuscated compared to the original code using *digital signatures* by finding similar pieces of text, i.e. for matching strings of words. We tried to eliminate the threat of using such a syntactic tools in decompilations by decompiling the original (clear) byte-code of each benchmark and the obfuscated version using the same decompiler; then we could apply our matching tool with a high confidence of producing accurate matching results (see also Section 7.3.4).

8.4.2 Construction Validity

Construction validity checks if our approach (theory) has any actual, real relation with observation (resilience of code obfuscation) using the dependent variable: percentage of failing to find the original code from the obfuscated one. It is used to describe the extent to which our measurements

describe the desired attribute we want to compute (code obfuscation security). We justify the choice of this factor, as a valid evaluation factor for successful deobfuscation, in Section 7.3.4.

It is well known that reverse engineers apply different types of methods to successfully deobfuscate obfuscated code; beside decompilers (including static analysis and reverse engineering tools) they apply dynamic analysis such as debuggers and profilers. We do not claim that our threat model using decompilation is the only model to evaluate the resilience of code obfuscation. Decompilation is still an immature technique compared to compilers and other disciplines in software engineering, as decompilers can fail sometimes even for clear unobfuscated code. Despite these limitations, decompilers are still an important tool in the hands of reverse engineers to attack obfuscated code. In this experiment, we select three Java decompilers that are subject to an empirical validation for their effectiveness [HD09], in order to eliminate any bias relying on one decompiler.

8.4.3 Conclusion Validity

Threats to conclusion validity address the issues that affect the ability to draw the correct conclusion about the relation between our treatment (our proposed metric) and the outcome (resilience of code obfuscation). We justify the choice of selecting the statistical methods in Section 7.5. We also find a statistical significance between our proposed metrics, and decompilation. The conclusions have been drawn based on objective statistical tests; we have adopted non-parametric tests (such as Mann-Whitney and Spearman Rank Correlation) that do no make assumptions on the normal distribution of data. We analyse and study the multi-linear regression models using some diagnostic plots for error rate (regression models), and p-value for hypotheses testing.

8.4.4 External Validity

External validity investigates the extent to which we can generalise what we learn from our measurements to other (similar) disciplines or other programming paradigms. Our proposed metrics can be applied to any programming paradigm, with no restriction. However, since our regression model estimates the security in code obfuscation, it is only applicable to Java byte-code and cannot be generalised to all programming languages as in [CALO94] for Maintenance Index (*MI*). It will require more extensive empirical and experimental efforts to validate this result over benchmarks of different programming paradigms, which can be reserved for future work.

8.5 Summary

The empirical results demonstrate the possibility of using an Algorithmic Information Theory approach to measure the quality of code obfuscation. Based on Kolmogorov complexity and data compression, we establish consistent metrics for software protection without having to rely on classical software complexity metrics, to reason about the amount of security added by each obfuscation technique. We empirically validated the usefulness of the proposed metrics (*NCD*, π_U and *NC*). The results shows that all the proposed measures are positively correlated with the percentage of code obfuscation resilience to an attacker using decompilers, which indicates a positive relationship with the obfuscation resilience factor (percentage of failing to retrieve the original clear code), yet at different level of correlations. The *NCD* offers the highest correlation, the π_U shows a medium correlation, where the *NC* scores the lowest. We also construct a multi-linear regression model, using the proposed metrics, to estimate Java programs resilience to decompilation attacks.

We used the proposed metrics to study the quality of code obfuscation. We find in the most of cases the effectiveness of an attacker (using a decompiler) is greatly reduced by the level of noise and irregularities introduced by obfuscation techniques. The more code is lifted from the obfuscated `jar` files using decompiler, the less the degree of similarity distance (using *NCD*), Unintelligibility Index (π_U) and normalised Kolmogorov complexity (using *NC*). The analysis of the results shows the importance of defining a clear attack model, which has to be taken into account when we apply the proposed measures. In general, we see from the obtained results that Dasho's obfuscation techniques outperformed Sandmark's obfuscation techniques using all proposed measures. Among all decompliers we find that *Jode* is the best decompiler according to all proposed metrics. Layout obfuscation techniques exceed Data and Control-Flow obfuscation techniques using *NCD*. On the other hand Control-Flow obfuscation techniques score the best values according to π_U. In case of *NC*, Layout, Control-Flow, and Data obfuscation techniques achieve similar results.

9 Conclusion

Code obfuscation presents an effective and promising protection mechanism for software intellectual property; it can be extended to solve may problems that cryptography has not yet addressed. However, code obfuscation is as yet in its infancy, and very few theoretical investigations exist that reason about its security. Moreover, the early theoretical work by Barak et al. showed that it is impossible to find a general purpose obfuscator that can efficiently obfuscate programs, and secure programs according to the virtual black-box security model. On the other hand, the recent theoretical advances by Garg et al. using indistinguishably obfuscation, have shown the possibility of constructing secure obfuscation based on linear maps, Multilinear Jigsaw Puzzles, that satisfy indistinguishability obfuscation. Despite that result, there are no theoretical or practical frameworks that can reason about the security of current obfuscation techniques. Furthermore, it is an open problem whether there exist quantitative metrics that can measure the quality of code obfuscation.

In this thesis, we attempted to tackle these challenges by following two research methods. We pursue a theoretical and formal approach to address the lack of theoretical foundations for the security of code obfuscation, and find quantitative metrics that can measure and certify obfuscated programs. Then we applied an experimental validation approach to evaluate and test whether our proposed model (theory and metrics) is empirically sound. We proposed a novel approach that established a theoretical foundation for code obfuscation security which quantitatively captures the level of confusion that is added by code obfuscation.

The main idea of this approach is to apply Algorithmic Information Theory (Kolmogorov complexity) and algorithmic mutual information to build security foundations for code obfuscation, and to model adversaries with deobfuscation capabilities. First, we defined code obfuscation based on the notion of Unintelligibility; the rationale behind this definition is that an obfuscated program must be more difficult to understand than the original program. Then, we showed that in some cases unintelligibility in not sufficient to reason about code obfuscation security, which led to a

further definition for code obfuscation security that is based on algorithmic mutual information. This definition allows us to reveal a negligible amount of information about the original program to an adversary.

We compared our security definition with the famous impossibility result of code obfuscation, and we showed that our proposed security model differs substantially from the virtual black-box obfuscation model that led to this negative result, in the sense that our definition is a less stringent form of obfuscation rather than a weak form of black-box obfuscation. We assume the functionality of an obfuscated program to be almost completely known and available to an adversary, and only require hiding the implementation rather than the functionality itself. This approach to obfuscation is very practical and pragmatic, especially for software protection obfuscation. We further compared our security model with indistinguishability obfuscation, and argued that with indistinguishable obfuscation it is very difficult to provide a guarantee about what obfuscation hides. We also showed, according to our proposed definition of security and under reasonable conditions, code obfuscation is secure.

We investigated the security of two main approaches to obfuscated code in software, *encoding* and *hiding*, at the subprogram level. We studied the security of combining different obfuscation techniques, which is essential for dynamic obfuscation, and malware design that employs different obfuscations during runtime to avoid detection.

We model adversaries with deobfuscation capabilities, and compute the attack outcome (success) algorithmic mutual information. We theoretically showed that Kolmogorov complexity is a valid metric to measure software. Then we derived a comprehensive set of quantitative metrics that are approximated by lossless compression: unintelligibility index, normalised Kolmogorov complexity, normalised Compression distance, and code obfuscation stealth, to measure the quality of code obfuscation, and justify their usage in the light of our security model.

The proposed metrics were validated empirically to check whether they are effective and sound metrics to measure code obfuscation security. We obfuscated the SPECjvm2008 benchmark programs using two obfuscators: Sandmark, an open-source obfuscation suite, and Dasho, a commercial obfuscator. Then we applied three different decompilers, as a model of attack, to study the degree of resilience exhibited by each obfuscated programs. The results showed that all the proposed measures are positively correlated with the percentage of code obfuscation resilience to an attacker using decompilers, which indicate a positive relationship with obfuscation resilience

factor (the percentage of the clear code that was not recovered); however, at different levels of correlations. Furthermore, the results shed light on the danger of relying on absolute metrics as a means to predict the obfuscation resilience, without taking into account the attacker itself when measuring. We also found that the initial complexity of a clear program affects the quality of the obfuscated version. Therefore, the defender has to decide whether the clear code needs to be obfuscated or not, and which features to obfuscate.

To the best of our knowledge, our model presents a new practical approach to measure, quantitatively, the security of the current state of art obfuscation techniques, without having to rely on classical complexity metrics as in [CCFB14, CTL97]. The results of this thesis partially complement the experimental work of Jbara et al. [JF14] on program comprehension and understanding using compression, by providing rigorous theoretical foundation and metric validation, which justify the use of Kolmogorov complexity and compression, as the upper bound approximation for Kolmogorov complexity, to measure unintelligibility in code obfuscation. However, we advance on this theory and use algorithmic mutual information as the basis for code obfuscation security.

9.1 Future Work

Our proposed theoretical security, the adversary model, and the proposed metrics are by no means complete to predict and compute the security of code obfuscation, and we do not claim it is the best achievable overall. It only presents a milestone towards paving the way and inspire security researchers into adapting, and building a sufficient and complete metrics to quantify code obfuscation security. Towards achieving this important goal, we propose the following research directions.

Characterise the security of particular obfuscation technique. We are planning to study and characterise the security of particular obfuscation techniques, and to analyse more carefully the scenario of active adversaries that equipped with dynamic analysis tools. For example, in Section 4.4 we studied the security of combining multiple obfuscation techniques; however, we did not cover the case of self-modified obfuscated code that is used to counter static-analysis attacks. In this technique, the program is in constant flux of changing, and an adversary attacks (typically using dynamic analysis tools) each version of the modified obfuscated code. S/he may obtain *partial* information on every single instance of multiple

obfuscation, and tries to combine the partial information in order to construct the original clear code. That requires a security analysis for every single instance of modified obfuscated program separately.

Experimental evaluation for dynamic obfuscation technique. The proposed metrics were validated empirically on a set of Java source and byte code of SPECjvm2008 programs. We are interested in extending our validation framework to cover other programming languages such as *C, C++, JavaScript* and *assembly language*.

In our experiment, we used an attack model where the adversary can only use static analysis techniques; the metrics that we proposed do not certify code obfuscations against dynamic analysis tools such as profiling, debugging and dynamic slicers.

Experimental work to validate stealth in code obfuscation. One of the main problems with measuring stealth is that it is context dependant. In Section 6.4.5 we propose a modified version of Information distance that measures the intrinsic and extrinsic stealth in obfuscated programs, and eliminates the dependency factor. However, we did not provide empirical evidence about the validity of this metric. We are planning to design experiments, in a similar way to Chapter 7 and 8, to measure and validate the stealth measure, and to test its effectiveness in detecting and distinguishing obfuscated code from clear code. Another interesting research direction, would be to apply the proposed stealth measure to classify and detect malware in a similar way to [ABCD15].

Kolmogorov complexity to measure program semantics Compression is used as an upper bound approximation for Kolmogorov complexity. So far we present our theoretical foundation and metrics for code obfuscation security based on binary strings; a very important question that can rise here is whether can we apply our approach on programs semantics instead of binary strings. Giacobazzi et al. [GR97] propose the notion of Domain Compression in abstract interpretation theory (see Section 3.6.2) as a refinement of a finite abstract domain, and showed that is possible to simplify abstract domains through abstract compression. We envision that this approach could be the way to reason about Kolmogorov complexity of programs based on their semantics properties. If the concrete semantics of an obfuscated program are abstracted and then compressed, it could be a way to measure the complexity of the programs' semantics in obfuscated code. Moreover, it can be used to study the security

of more sophisticated obfuscation techniques that require a semantic-reasoning approach to deobfuscate [CLD11b] as in virtual obfuscations (see Section 3.6.4).

Explore the relation between Shannon Entropy and Kolmogorov complexity. Shannon Entropy is asymptotically equal to Kolmogorov complexity, as Shannon entropy is the expected value of Kolmogorov complexity (see Section 2.11). Despite that relation, it is unclear whether Shannon Entropy can equivalently address the security of code obfuscation as with Kolmogorov complexity, and under which conditions; an interesting research direction that is worth investigation, is to study code obfuscation security in the context of Classical Information Theory.

Towards a unified theory for software protection. Finally, Algorithmic Information Theory can be the tool to study security in many open problems in software protection such as software watermarking, software birthmark, and software temper-proofing. We envisage that Algorithmic Information Theory is a candidate for building a unified theory for software protection, that could lead to new, effective, provably secure, and practical, software protection techniques.

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[ABCD15] Nadia Alshahwan, Earl T Barr, David Clark, and George Danezis. Detecting malware with information complexity. *arXiv preprint arXiv:1502.07661*, 2015.

[AMDS+07] Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. Program obfuscation: a quantitative approach. In *Proceedings of the 2007 ACM workshop on Quality of protection*, pages 15–20. ACM, 2007.

[App02] Andrew W Appel. Deobfuscation is in NP. *Symposium A Quarterly Journal In Modern Foreign Literatures*, pages 1–2, 2002.

[Arb02] Genevive Arboit. A method for watermarking java programs via opaque predicates. In *In Proc. Int. Conf. Electronic Commerce Research (ICECR-5)*, 2002.

[Arb11] Tom Arbuckle. Studying software evolution using artefacts' shared information content. *Sci. Comput. Program.*, 76(12):1078–1097, December 2011.

[AS15] Gilad Asharov and Gil Segev. Limits on the power of indistinguishability obfuscation and functional encryption. Cryptology ePrint Archive, Report 2015/341, 2015. `http://eprint.iacr.org/`.

[Auc96] David Aucsmith. Tamper resistant software: An implementation. In *Proceedings of the First International Workshop on Information Hiding*, pages 317–333, London, UK, 1996. Springer-Verlag.

[BCC+14] Nir Bitansky, Ran Canetti, Henry Cohn, Shafi Goldwasser, Yael Tauman Kalai, Omer Paneth, and Alon Rosen. The impossibility of obfuscation with auxiliary input or a universal simulator. In *Advances in Cryptology–CRYPTO 2014*, pages 71–89. Springer, 2014.

[BF07] Philippe Beaucamps and Eric Filiol. On the possibility of practically obfuscating programs towards a unified perspective of code protection. *Journal in Computer Virology*, 3(1):3–21, 2007.

[BGI+01] 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.

[BGI+12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6:1–6:48, May 2012.

[BGK+14] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. In*Advances in Cryptology–EUROCRYPT 2014*, pages 221–238. Springer, 2014.

[BGM+98] Charles H. Bennett, Peter Gacs, Senior Member, Ming Li, Paul M. B. Vitányi, and Wojciech H. Zurek. Information distance. *IEEE Transactions on Information Theory*, 44:1407–1423, 1998.

[BMB96] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. Property-based software engineering measurement. *IEEE Trans. Software Eng.*, 22(1):68–86, 1996.

[BMRP09] Guillaume Bonfante, Jean-Yves Marion, and Daniel Reynaud-Plantey. A Computability Perspective on Self-Modifying Programs. In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 231–239. IEEE, 2009.

[BOKP15] Sebastian Banescu, Martín Ochoa, Nils Kunze, and Alexander Pretschner. Idea: Benchmarking indistinguishability obfuscation–a candidate implementation. In *Engineering Secure Software and Systems*, pages 149–156. Springer, 2015.

[BW94] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In *DIGITAL SRC RESEARCH REPORT*. Citeseer, 1994.

[CALO94] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, August 1994.

[CAO05] Manuel Cebrián, Manuel Alfonseca, and Alfonso Ortega. Common pitfalls using normalized compression distance: what to watch out for in a compressor. *Communications in Information and Systems*, 5:367–384, 2005.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[CC02] Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. *ACM SIGPLAN Notices*, 37(1):178–190, January 2002.

[CCFB14] 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.

[CGJZ01] Stanley Chow, Yuan Xiang Gu, Harold Johnson, and Vladimir A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. pages 144–155, 2001.

[Cha66] Gregory J. Chaitin. On the length of programs for computing finite binary sequences. *Journal of the ACM*, 13:547–569, 1966.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[CIW84] John G. Cleary, Ian, and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32:396–402, 1984.

[CLD11a] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 275–284. ACM, 2011.

[CLD11b] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 275–284. ACM, 2011.

[CMT15] Testwell CMTJava: Complexity measures tool for Java. `http://www.testwell.fi/cmtjdesc.html`, 2015.

[CN09] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, August 2009.

[Cou96] Patrick Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, June 1996.

[Cou97] Patrick Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation (Extended Abstract). *Electronic Notes in Theoretical Computer Science*, 6:77–102, 1997.

[CPN+09] 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.

[CT06] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2006.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[CV05] Rudi Cilibrasi and Paul M. B. Vitnyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51:1523–1545, 2005.

[CX12] Silvio Cesare and Yang Xiang. *Software Similarity and Classification*. Springer, 2012.

[das15] Dasho. `http://www.preemptive.com/products/dasho`, 2015.

[DH08] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. *SIGPLAN Not.*, 43(10):313–328, October 2008.

[DMT07] Stephen Drape, Anirban Majumdar, and Clark Thomborson. Slicing aided design of obfuscating transforms. In *Computer and Information Science, 2007. ICIS 2007. 6th IEEE/ACIS International Conference on*, pages 1019–1024. IEEE, 2007.

[Dra10] Stephen Drape. Intellectual property protection using obfuscation. Technical Report RR-10-02, March 2010.

[DREB98] Willem-Paul De Roever, Kai Engelhardt, and Karl-Heinz Buth. *Data refinement: model-oriented proof methods and their comparison*. Cambridge University Press, 1998.

[FS98] Ronald Fagin and Larry Stockmeyer. Relaxing the triangle inequality in pattern matching. *International Journal of Computer Vision*, 30(3):219–231, 1998.

[FWWH11] Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. Multi-stage binary code obfuscation using improved virtual machine. In *Information Security*, pages 168–181. Springer, 2011.

[Gác74] Peter Gács. On the symmetry of algorithmic information. *Soviet Math. Dokl*, 15:1477–1480, 1974.

[GCK05] Jonathon T Giffin, Mihai Christodorescu, and Louis Kruger. Strengthening software self-checksumming via self-modifying code. In *Computer Security Applications Conference, 21st Annual*, pages 10–pp. IEEE, 2005.

[GGH+13] Shelly Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Anant Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 40–49. IEEE, 2013.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[GR97] Roberto Giacobazzi and Francesco Ranzato. Refining and compressing abstract domains. In *Automata, Languages and Programming*, pages 771–781. Springer, 1997.

[GR07] Shafi Goldwasser and Guy N Rothblum. On best-possible obfuscation. In*Theory of Cryptography*, pages 194–213. Springer, 2007.

[GTV01] Péter Gács, John T. Tromp, and Paul M.B. Vitányi. Algorithmic statistics. *IEEE Transactions on Information Theory*, 47:2443–2463, 2001.

[GV04] Peter Grünwald and Paul M. B. Vitányi. Shannon Information and Kolmogorov Complexity. *CoRR*, cs.IT/0410002, 2004.

[GZD11] Nicolas Gauvrit, Hector Zenil, and Jean-Paul Delahaye. Assessing cognitive randomness: A Kolmogorov complexity approach. *CoRR*, abs/1106.3059, 2011.

[Had00] Satoshi Hada. Zero-knowledge and code obfuscation. In *Advances in Cryptology-ASIACRYPT 2000*, pages 443–457. Springer, 2000.

[Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series).* Elsevier Science Inc., New York, NY, USA, 1977.

[HD09] James Hamilton and Sebastian Danicic. An evaluation of current java bytecode decompilers. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 129–136. IEEE, 2009.

[HK81] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *Software Engineering, IEEE Transactions on*, (5):510–518, 1981.

[HM81] Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *ACM SIGPLAN Notices*, 16(3):63–74, March 1981.

[Hor97] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997.

[HW09] Hsin-Yi Tsai, Yu-Lun Huang and David Wagner. A Graph Approach to Quantitative Analysis of Control-Flow Obfuscating Transformations. *IEEE Transactions on Information Forensics and Security*, 4(2):257–267, June 2009.

[Ird13] Sustainable device security: Breaking the hacker business model with software security, Irdeto white paper. `http://irdeto.com/documents/Collateral/wp_sustainable_device_security_en.pdf`, 2013.

[JAD15] JAD. `http//varaneckas.com/jad/`, 2015.

[JD15] JD. `http://java.decompiler.free.fr`, 2015.

[JF14] Ahmad Jbara and Dror G Feitelson. On the effect of code regularity on comprehension. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 189–200. ACM, 2014.

[Jod15] Jode. `http://jode.sourceforge.net/`, 2015.

[Kin12] Johannes Kinder. Towards static analysis of virtualization-obfuscated binaries. In *Proc. 19th Working Conf. Reverse Engineering (WCRE 2012)*, pages 61–70. IEEE, 2012.

[KJ04] Steven R Kirk and Samantha Jenkins. Information theory-based software metrics and obfuscation. *Journal of Systems and Software*, 72(2):179–186, 2004.

[KKM88] David G. Kleinbaum, L. Lawrence Kupper, and Keith E. Muller. *Applied Regression Analysis and Other Multivariable Methods*. PWS Publishing Co., Boston, MA, USA, 1988.

[Kle64] Stephen C. Kleene. *Introduction to metamathematics*. Bibliotheca mathematica. North-Holland Pub. Co., 1964.

[KLST71] David Krantz, Duncan Luce, Patrick Suppes, and Amos Tversky. *Foundations of Measurement, Vol. I: Additive and Polynomial Representations*. New York Academic Press, 1971.

[KMNM03] Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto. Exploiting self-modification mechanism for program protection. In *Computer Software*

and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International, pages 170–179. IEEE, 2003.

[Kol65] Andrey N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems in Information Transmission*, 1:1–7, 1965.

[Kra49] Leon G. Kraft. A device for quantizing, grouping, and coding amplitude modulated pulses. M.Sc. Thesis, Dept. of Electrical Engineering, MIT, Cambridge, Mass., 1949.

[KSLT89] David H. Krantz, Patrick Suppes, R.Duncan Luce, and A. Tversky. *Foundations of measurement, Vol. II: Geometrical, threshold, and probabilistic representations*. Foundations of Measurement. Academic Press, 1989.

[KY96] 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.

[Lan92] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[Lat97] James I. Lathrop. Compression depth and the behavior of cellular automata. *Complex Systems*, 1997.

[LCL+04] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul M. B. Vit ányi. The similarity metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264, 2004.

[LD03] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[LG07] Alberto Leon-Garcia. *Probability and Random Processes For EE's (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2007.

[LR96] Sophie Laplante and John Rogers. Indistinguishability, 1996.

[LV08] Ming Li and Paul M.B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Publishing Company, Incorporated, 3 edition, 2008.

[LXX+09] Jian Li, J. Xu, Ming Xu, H.L. Zhao, and Ning Zheng. Malware obfuscation measuring via evolutionary similarity. In *Future Information Networks, 2009. ICFIN 2009. First International Conference on*, pages 197–200. IEEE, 2009.

[MAM+05] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. Software protection through dynamic code mutation. In *Information Security Applications*, pages 194–206. Springer, 2005.

[Mat15] Matlab. `http://www.mathworks.co.uk/products/matlab/`, 2015.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[MK93] John C. Munson and Taghi M. Khoshgoftaar. Measurement of data structure complexity. *Journal of Systems and Software*, 20(3):217–225, 1993.

[MKP11] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 2011.

[MTD06] Anirban Majumdar, Clark D. Thomborson, and Stephen Drape. A survey of control-flow obfuscations. pages 353–356, 2006.

[Muc02] Andrei A. Muchnik. Conditional complexity and codes. *Theor. Comput. Sci.*, 271(1-2):97–109, 2002.

[Muc11] Andrej Muchnik. Kolmogorov complexity and cryptography. *CoRR*, abs/1106.5433, 2011.

[NNH15] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[Ost77] Leon J. Osterweil. The detection of executable program paths through static data flow analysis. *Department of Computer Science, University of Colorado at Boulder*, 1977.

[PAK98] Fabien AP Petitcolas, Ross J Anderson, and Markus G Kuhn. Attacks on copyright marking systems. In *Information Hiding*, volume 1525, pages 218–238. Springer, 1998.

[PG09] Mila Dalla Preda and Roberto Giacobazzi. Semantics-based code obfuscation by abstract interpretation. *Journal of Computer Security*, 17(6):855–908, 2009.

[Pin07] Alexandre Miranda Pinto. *Application of Kolmogorov Complexity to Cryptography*. PhD thesis, Departamento de Ciências de Computadores, Faculdade de Ciências da Universidade do Porto, 2007.

[Pre07] Milla Dalla Preda. *Code Obfuscation and Malware Detection by Abstract Interpretation*. PhD thesis, Dipartimento di Informatica, Universita' di Verona, 2007.

[Ric53] H G Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[Rob15] Borut Robič. *The Foundations of Computability Theory*. Springer-Verlag Berlin Heidelberg, 1st edition, 2015.

[Rol09] Rolf Rolles. Unpacking virtualization obfuscators, proceedings of the 3rd usenix conference on offensive technologies. pages 1–1, 2009.

[Sal06] David Salomon. *Data Compression: The Complete Reference*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[san15] Sandmark. `http://sandmark.cs.arizona.edu`, 2015.

[SGA07] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948.

[She82] Alexander Shen. Axiomatic description of the entropy notion for finite objects. *VIII All-USSR Conference (Logika i metodologija nauki),Vilnjus*, pages 104 – 105, 1982. The paper in Russian.

[She07] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 4 edition, 2007.

[she15] Sherlock. `http://www2.warwick.ac.uk/fac/sci/dcs/research/ias/software/sherlock/`, 2015.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 3rd edition, 2013.

[SL93] Patricia Snyder and Stephen Lawson. Evaluating results using corrected and uncorrected effect size estimates. *The Journal of Experimental Education*, 61(4):334–349, 1993.

[SL12] G.A.F. Seber and A.J. Lee. *Linear Regression Analysis*. Wiley Series in Probability and Statistics. Wiley, 2012.

[Sol64] Ray J. Solomonoff. A formal theory of inductive inference. part i. *Information and Control*, 7(1):1–22, 1964.

[Spe08] Specjvm2008. `https://www.spec.org/jvm2008/`, 2008.

[Ste04] Stephen Drape. Obfuscation of Abstract Data Types. DPhil thesis, Oxford University Computing Laboratory. 2004.

[Sti05] David Stirzaker. *Stochastic processes and models*. Oxford University Press, New York, 2005.

[STZDG12] Fernando Soler-Toscano, Hector Zenil, Jean-Paul Delahaye, and Nicolas Gauvrit. Calculating Kolmogorov complexity from the output frequency distributions of small turing machines. *CoRR*, abs/1211.1302, 2012.

[STZDG14] Fernando Soler-Toscano, Hector Zenil, Jean-Paul Delahaye, and Nicolas Gauvrit. Calculating kolmogorov complexity from the output frequency distributions of small turing machines. *PloS one*, 9(5):e96223, 2014.

[SUV14] A. Shen, V. Uspensky, and N. Vereshchagin. *Kolmogorov complexity and algorithmic randomness*. MCCME Publishing house, 2014.

[Tav11] Antoine Taveneaux. Towards an axiomatic system for kolmogorov complexity. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:14, 2011.

[Tur36] Alan Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. page 42. Proceedings of the London Mathematical Society, Series 2, 1936.

[TZ92] Jianhui Tian and Marvin V. Zelkowitz. A formal program complexity model and its application. *Journal of Systems and Software*, 17(3):253 – 266, 1992.

[UDM05] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering, WCRE 2005, Pittsburgh, PA, USA, November 7-11, 2005*, pages 45–54, 2005.

[vB15] Steffen van Bakel. Lecture notes in discrete mathematics, part i, Autumn 2015.

[WDHK01] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. Protection of software-based survivability mechanisms. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 193–202. IEEE, 2001.

[Wey88] Elaine J Weyuker. Evaluating software complexity measures. *Software Engineering, IEEE Transactions on*, 14(9):1357–1365, 1988.

[Wol95] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[WRH+00] Claes Wohline, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[YJWD15] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 674–691. IEEE, 2015.

[ZHZ+07] Xian Zhang, Yu Hao, Xiaoyan Zhu, Ming Li, and David R Cheriton. Information distance from a question to an answer. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining* , pages 874–883. ACM, 2007.

[ZHZ10] Xuesong Zhang, Fengling He, and Wanli Zuo. *Theory and practice of program obfuscation*. INTECH Open Access Publisher, 2010.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transaction on Information Theory*, 23(3):337–343, 1977.