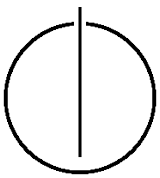
FAKULTÄT FÜR INFORMATIK

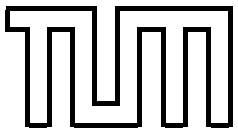
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation in Informatik

Characterizing the Strength of Software Obfuscation Against Automated Attacks

Sebastian-Emilian Bănescu





FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl XXII - Software Engineering

Characterizing the Strength of Software Obfuscation Against Automated Attacks

Sebastian-Emilian Bănescu

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Alfons Kemper, Ph.D.
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Alexander Pretschner
2.	Prof. Saumya Debray, Ph.D., University of Arizona, USA

Die Dissertation wurde am 28.04.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 29.06.2017 angenommen.

Acknowledgments

Motto: *In theory, theory and practice are the same, but in practice, they are different.*

When I embarked on the journey of pursuing a PhD, I was aware of how it would be like in theory. I have since realized that it was *different* in practice. However, throughout this journey I have learned a lot from the amazing people I met along the way. Therefore, I would like to use the following paragraphs to thank some of the people who have helped, supported and motivated me throughout my journey until this point.

Innumerable thanks to my supervisor Prof. Dr. Alexander Pretschner for giving me the opportunity to define and pursue my own PhD topic. Your invaluable support and feedback has guided me throughout the entire development of this thesis. I have learned so many important things from you over the years, like: being concise when sharing ideas, not being afraid to admit when I do not know something and being critical, but not over the top.

I would like to thank my second supervisor Prof. Dr. Saumya Debray for the kindness he showed me during our conversations and during my visit to Tucson. His feedback, encouragement and suggestions have helped me in the toughest period of my PhD. One important thing I have learned from you is to analyze things from a broader perspective.

Thank you to my mentor Assist. Prof. Dr. Martín Ochoa for his patience in listening to my half-baked ideas and steering me in the right direction when I was getting side-tracked with irrelevant details. From you, I learned how to focus, in order to get things done.

I am also grateful to Assoc. Prof. Dr. Christian Collberg, Assist. Prof. Dr. Vijay Ganesh and the Google engineers for their feedback and suggestions on the papers which we have collaborated on. Their points of view taught me to analyze ideas from different angles.

To my present and former colleagues at TU Munich – the list of names is too long, but you know who you are :) – many thanks for always cheering me up and encouraging me when my moral was down. From you girls and boys, I have learned how to enjoy lunch and coffee breaks at work, as well as how to collaborate with colleagues.

Thanks to all of the reviewers of this thesis and to the Bachelor and Master students who have helped me develop and improve my thesis.

I would also like to share my gratitude to my family and friends for their continuous support and encouragement. My parents have taught me many important lessons throughout my life, one of which was that *no one can steal the knowledge from your head, which makes education the most valuable asset that I can ever own.* Special thanks to my friend Radu, who sparked my interest in academic research, which led me to pursue a PhD.

Finally and most importantly, I would like to thank my wife Iulia for her endless love, support, encouragement and patience. Without all the sacrifices you have done for me, I would have not been able to successfully complete this journey. There is not enough paper in the world for me to express my gratitude, but I will try to show it to you every day.

Zusammenfassung

Softwareverschleierung bezeichnet eine Art der Transformationstechniken für Computercode, die von Software Entwicklern eingesetzt wird, um digitales Eigentum (z.B. proprietäre Algorithmen, geheime Schlüssel, etc.) gegen böswillige Endbenutzer zu schützen. In den letzten drei Jahrzehnten sind Dutzende Verschleierungstransformationen in der Literatur beschrieben worden. Einige Forscher haben sich darauf konzentriert, die Stärken dieser Transformationen gegen *semi-automatisierte Analysen* zu beschreiben und zu messen. Im Vergleich dazu wurden weniger Fortschritte darin gemacht, die Stärken dieser Transformationen gegen *voll-automatisierte Angriffen* zu beschreiben und zu messen. Der hauptsächliche Grund hierfür ist, dass die übliche Form der Softwaredistribution (auch "Software Monokultur" genannt), davon ausgeht, dass eine Anwendung einmal verschleiert wird und die gleiche ausführbare Datei an alle Endnutzer verteilt wird. Daher kann, falls ein böswilliger Endnutzer in der Lage ist die schützende Verschleierung zu überwinden, der gleiche Angriff bei allen anderen Endnutzern der gleichen Anwendung angewendet werden. Mit dem Aufkommen der Softwarediversität erhalten verschiedene Endnutzer der Software verschiedene, aber funktionsgleiche, Versionen des gleichen Programms, die mit verschiedenen Kombinationen von Transformationen verschleiert wurden. Idealerweise bedeutet das, dass ein und derselbe Angriff nicht mehr auf alle Endnutzer anwendbar ist und semi-automatisierte Analysen mit den Millionen verschiedenen Varianten eines Programms, die jeden Tag erscheinen, nicht mehr mithalten können. Softwarediversität wird in großen Maßen von Schadsoftware-Entwicklern eingesetzt, um nicht von Antiviren-Scannern entdeckt zu werden. Auch ist Softwarediversität mit Hilfe von Verschleierung als Abwehrmechanismus für gutartige Programme gegen Angriffe vorgeschlagen worden, die darauf abzielen, die Integrität der Anwendung durch Manipulationen am Computercode (engl. "Tampering"), das Einfügen von Computercode oder dem Wiederverwenden von Computercode zu verletzen. Nichtsdestotrotz stehen Fachleute dem Einsatz von Verschleierung zum Schutz ihrer eigenen Software skeptisch gegenüber. Dies liegt zum Teil daran, dass nicht klar ist, wie die Stärke von Software Verschleierung charakterisiert werden kann.

In dieser Dissertation entwickeln wir eine Grundstruktur, in der wir die automatische Analyse als Suchprobleme formalisieren, deren Komplexität von Eigenschaften der Software abhängt. Mit dieser Grundstruktur lassen sich die Stärken gegen Angriffe durch automatisierte Analyse beschreiben und vergleichen. Dies hilft Entwicklern dabei, die Verschleierungtransformationen auszuwählen, die die Eigenschaften der Software so verändern, dass Heuristiken nicht mehr angewendet werden können oder der benötigte Aufwand für die Suche aus wirtschaftlicher Sicht nicht mehr attraktiv ist. Wir stellen mehrere Fallstudien unter Einbezug von verschiedenen Software-Anwendungen, Verschleierungstransformationen und einem automatisierten Angriff bassierend auf symbolischer Ausführung vor, die unsere Hypothese untermauern. Mit Hilfe der Erkenntnisse unseres Ansatzes zur Charakterisierung der Stärken von Verschleierung, sind wir in der Lage den Stand der Technik bei Verschleierungstransformationen voranzutreiben.

Abstract

Software obfuscation is a category of code transformation techniques employed by software developers to protect digital assets (e.g. proprietary algorithms, secret keys, etc.), against malicious end users of their applications. Dozens of obfuscating code transformations have been published in the last three decades, and some researchers have focused on quantifying and characterizing the strength of these transformations against *human-assisted analysis*. Relatively fewer advances have been made in quantifying and characterizing the strength of these transformations against *automated analysis*. This is because the general software distribution model (also called the "software monoculture") assumes that an application is obfuscated once and the same executable code is distributed to all end-users. Therefore, if one malicious end-user is able to bypass the obfuscation which protects an application, then the same attack can be applied to all other end-users of the same application. However, with the dawn of "software diversity" different software end-users receive different instances of the same software, obfuscated using different combinations of code transformations. Ideally, this means that the same attack is no longer applicable to all end-users and human-assisted analysis no longer scales when millions of software instances appear every day. On the one hand, software diversity is heavily employed by malware developers, in order to evade detection by anti-virus engines. On the other hand, software diversity via code obfuscating transformations has been proposed as a defense mechanism of benign software against attacks aiming to violate the integrity of software behavior via code manipulation (i.e. tampering), code injection and code reuse. However, practitioners are skeptical about employing obfuscation for protection of their own software. This is, in part, due to the fact that it is not clear how to characterize the strength of software obfuscation.

In this thesis, we develop a framework for the characterization of software obfuscation strength against automated analysis attacks. We do this by formulating automated analysis as search problems, whose complexities depend on various software characteristics. These characteristics become apparent after an attack is formulated using our framework. This helps developers to choose obfuscating transformations, which change those software characteristics, such that heuristics are no longer applicable or to increase the search effort to an extent that it is no longer economically attractive. We present multiple experiments involving various software applications, obfuscating transformations and an automated attack based on symbolic execution, whose results support our hypothesis. Using the insights gained from this approach towards obfuscation strength characterization, we are able to improve the state of the art of obfuscating transformations.

Outline of the Thesis

Chapter 1: Introduction

This chapter presents an introduction to the topic and to the fundamental issues addressed by this thesis. It discusses context, motivation, goals and limitations of this work.

Chapter 2: Obfuscation in Theory

This chapter presents a brief overview of cryptographic obfuscation and why it is currently far from being practical. Parts of this chapter have previously appeared in a peer-reviewed publication [14], co-authored by the author of this thesis.

Chapter 3: Obfuscation in Practice

This chapter presents an overview of obfuscation and software diversity transformations employed in practice. It also describes challenges of building practical obfuscators. Parts of this chapter have also appeared in a publication [22], co-authored by the author of this thesis.

Chapter 4: Automated MATE Attacks

This chapter presents the main contribution of this thesis: a model for reasoning about obfuscation strength by representing different steps of all automated Man-At-The-End (MATE) attacks as search problems. Parts of this chapter have previously appeared in a peer-reviewed publication [21], co-authored by the author of this thesis.

Chapter 5: Code Obfuscation Against Symbolic Execution Attacks

This chapter presents a characterization of automated symbolic execution attacks based on the model from Chapter 4. These characteristics are used to reason about and compare the resilience of a subset of obfuscation transformations from Chapter 3. Parts of this chapter have been published in a peer-reviewed publication [18] co-authored by the author of this thesis.

Chapter 6: Predicting Cost of Symbolic Execution Attacks on Obfuscated Code

This chapter presents a framework for predicting the time required by a successful symbolic execution attack, on obfuscated programs. The framework requires the metrics stated in Chapter 5. Parts of this chapter have appeared in a publication [19] co-authored by the author of this thesis.

Chapter 7: Improving Obfuscation Transformations Against Symbolic Execution

This chapter presents novel obfuscation transformations, which aim to raise the bar against symbolic execution attacks. These transformations specifically target the program charac-

teristics derived using the search model from Chapter 4. Parts of this chapter have been published in two peer-reviewed publications [18, 20] co-authored by the author of this thesis.

CHAPTER 8: RELATED WORK

This chapter presents related work in the sub-field of obfuscation strength evaluation and alternative solutions to software obfuscation and diversity for the purpose of software protection. Parts of this chapter have been published in peer-reviewed publications [21, 18, 17, 112] co-authored by the author of this thesis.

CHAPTER 9: CONCLUSIONS

This chapter first presents a summary of what has been done throughout the chapters of this thesis. Subsequently, we state the results of the thesis and the lessons learned during the development of this work. Afterwards, we discuss limitations and avenues for future work.

N.B.: *Multiple chapters of this dissertation are based on different publications authored or co-authored by the author of this dissertation. Such publications are mentioned in the short descriptions above. Due to the obvious content overlapping, quotes from such publications within the respective chapters are not marked explicitly.*

Contents

Part I.

Introduction and Background

1. Introduction

The Man-In-The-Middle (MITM) attacker model, formalized by Dolev-Yao in the early 1980s [72], has become the *de facto* standard in research papers dealing with secure communication. The goals of a MITM attacker include violating the confidentiality and integrity of information in transit between two trusted parties. A MITM attacker is an external third party, who does not have direct access to any internal states of the trusted parties. Originally, the MITM could eavesdrop on the communication between two trusted parties, hence, confidentiality was the main concern. MITM actions were extended to tampering with messages or impersonating a trusted party, raising the issues of integrity and authentication.

Research in the filed of security and cryptography, has led to mature protocols which are in wide-spread use since the 1990s and can withstand MITM attacks. On the other hand, the Man-At-The-End (MATE) attacker model [54], assumes that the attacker has (limited) control of one end, of a two-party interaction, e.g. the MATE is the end-user of an application developed by another party. The goals of a MATE attacker include violating the confidentiality of algorithms or other data inside of a software program and/or the integrity of software behavior as intended by the developer. Practically, any device under the control of an end-user (e.g. PC, TV, game console, mobile device, smart meter, etc.), running proprietary software is exposed to MATE attacks. The adversary is no longer a third party, in between two trusted parties, but rather one of them with physical, local or remote access to the target device.

A model of the MATE attacker capabilities, akin to the degree of formalization of the MITM attacker, is still missing from scientific literature. However, MATE attackers are assumed to be extremely powerful. They can examine software both statically using manual or automatic static analysis, or dynamically using state of the art software decompilers and debuggers [142]. Shamir *et al.* [189], present a MATE attack, which can retrieve a secret key used by a black-box cryptographic primitive to protect the system, if it is stored somewhere in non-/volatile memory. Moreover, the memory state can be inspected or modified during program execution and CPU or external library calls can be intercepted (forwarded or dropped) [225]. Software behavior modifications can also be performed by the MATE attacker by tampering with instructions (code) and data values directly on the program binary or after they are loaded in memory. The MATE attacker can even simulate the hardware platform on which software is running and alter or observe all information during software operation [53]. The only remaining line of defense in case of MATE attacks, is to increase the complexity of an implementation to such an extent that it becomes economically unattractive to perform an attack [53].

1.1. Benefits of Software Obfuscation

MATE attacks have raised the need for software protection mechanisms. Several techniques for software protection have emerged over the last two decades. The implementation of these different protection techniques can be done using: only software, software running on trusted hardware and/or software communicating with a trusted remote party (server). Using software-only protection techniques is the most attractive idea since it does not restrict the number users to those who have trusted hardware and eliminates the costs of setting up and maintaining a trusted remote server.

Implementations of such software-only protection mechanisms such as those offered by *Irdeto Cloakware* [115], *Arxan* [9], and *whiteCryption* [114] started being integrated in commercial products in the late 1990s when software vendors realized that a significant amount of end-users would rather crack their software, than buy a license for it [54]. Moreover, these cracks were applicable to all copies of that software and were easily distributed to other users. Therefore, they caused a loss in potential revenue for the software vendors. Nowadays, software protection is still heavily employed by malware developers, Digital Rights Management (DRM) systems, mobile applications, etc. Yet little is known about the how we can characterize the strength of such protection mechanisms. This leads to skepticism about any claimed security guarantees and to slow progress in the field of software protection.

Falcarin *et al.* [80], put the existing software-only protection techniques into four categories:

- *Obfuscation*, which thwarts reverse-engineering attacks by concealing its logic, data and identifiers.
- *Tamper-proofing*, which protects the integrity of software.
- *Watermarking*, which is used to trace back the original owner of unauthorized software copies.
- *Birthmarking*, which is used to determine code that has been copied from one program and used illegally in another program.

Given the large difference of attacker goals for each of the four categories of software protection techniques, in this thesis we choose to focus on *obfuscation*. Obfuscation is also divided in two major areas of research: (1) cryptographic obfuscation and (2) practical code obfuscation. Cryptographic obfuscation offers concrete security guarantees, nevertheless, they are currently far from being practical [25]. In this thesis we will briefly discuss the practical issues behind cryptographic obfuscation, however, the main focus will be on practical code obfuscation, which we will refer to simply as *obfuscation*. Obfuscation consists of software transformations at the level of source code, intermediate representation and/or native code, which aim to hide sensitive information available in the software application, from MATE attackers. Such sensitive information includes but is not limited to: the

algorithm performed by the software, the location of instructions which perform a certain functionality (e.g. decryption of a media stream, integrity checks, etc.), metadata (properties) of the program (e.g. whether it is malicious or not, which function it performs, which tools have been used to obfuscate the program, etc.) and confidential data (e.g. hard-coded keys, passwords, IP addresses, etc.).

Unfortunately, obfuscation cannot withstand a MATE attacker for an indeterminate period of time. History has repeatedly shown that given the right motivation a MATE attacker will be able to circumvent the obfuscation-based protection of a particular binary program [197]. This is particularly dangerous because of the current software development and distribution model called *software monoculture*, where all end-users receive a copy of the same binary for each software application. In software monocultures attacks are developed once and can subsequently be executed on all other software copies, running on systems of other end-users. Software diversity aims to decrease the applicability of MATE attacks on software by creating syntactically diverse, but functionally equivalent instances of one software program [86]. This does not stop the MATE from executing an attack against one particular software instance, and then create a tool which automatically applies this attack. Nevertheless, it increases the chances that the attack tool will not work against other instances of the same software, which eliminates the economical attractiveness of the attack tool.

Analogy with Cryptography Commercial obfuscation developers often keep the obfuscation algorithm and/or the details of its implementation secret, violating Kerckhoff's principle which states that a system should be secure even if everything about the system, except the key, is public knowledge [127]. Therefore, obfuscation is often associated with the term *security by obscurity*. However, by combining the ideas of software diversity and obfuscation we can move away from *security by obscurity*, towards something similar to cryptography, where only the key is kept secret, not the algorithm.

Similarly to cryptographic ciphers, the input configuration (e.g. sequence of obfuscation transformations, their parameters and random seeds) to a software obfuscation engine can be seen as a randomly chosen key, which characterizes the output of the engine in a unique way. We can attain software diversity by using different keys to protect different instances of the same software application. Ideally, this forces a MATE attacker to invest a similar amount of effort for attacking each different instance of the same software, similar to a cryptanalysis attack on ciphertexts encrypted with different keys. Therefore, it would be safe to make the obfuscation algorithms public and only protect the random key, akin to cryptographic ciphers. In this thesis we will always assume that the attacker has full knowledge about the implementation of the obfuscation transformations applied to the programs being attacked, nonetheless, s/he does not know the input configuration used for obfuscating that program.

As opposed to cryptographic ciphers, a successful MATE attack against an obfuscated application does not require recovering the secret key (except for metadata recovery at-

tacks [179] where the goal of the attacker is to recover the key). This is because obfuscation transformations (generally) do not have an inverse transformation, such as encryption and decryption. The reason why obfuscation transformations do not have an inverse is that many such transformations (similarly to compiler optimizations), destroy information about symbol names, comments, control flow, etc. Often this information cannot be recovered automatically by an inverse transformation. However, a MATE attacker's goal may be different from recovering the original (unobfuscated) version of the program, e.g. bypassing a check does not require recovering the original program, it just requires finding the location of the check in the obfuscated code and disabling it. Section 1.2 describes the capabilities and goals of the attacker which this thesis will focus on.

1.2. Attacker Model

Characterizing the strength of obfuscation against all MATE attacks is challenging, since this depends on: the goal of the attacker, the degree of knowledge of the attacker and the techniques and tools the attacker uses. The goals of the MATE attacker that this thesis will focus on are:

- Recovering hidden data (e.g. a password), from an obfuscated program.
- Exploring all executable code of an obfuscated program, e.g. in order to locate integrity checks or trigger conditions.

Moreover, in this thesis we focus on MATE attackers in the context of software diversity, where MATE attackers are not successful if they can achieve their goal on a single software instance. In the context of software diversity MATE attackers are only successful if they can automate their attacks such that they are applicable to all (or a majority of) obfuscated instances of a given software. If we make another analogy with cryptography here, then our attacker model is similar to the *ciphertext-only attack*, where the attacker is assumed to have access only to the ciphertext and no access to the corresponding plaintext. One interesting observation is that MATE attackers are not always malicious, they can also be benign. In the following paragraphs we present two scenarios where MATE attackers employ automated attacks for malicious and benign reasons, respectively.

Malicious MATE Malicious MATE attackers perform (often illegal) attacks which cause monetary loss for software vendors and/or end-users. One example of automated attacks is called *code patching*. Code patching modifies the code of a program (statically and/or dynamically) in order to change the input-output (IO) behavior of that program. Starting from the late 2000s some organizations started to automate such code patching attacks targeting popular applications (e.g. web browsers) in order to change their behavior in a way that would bring financial gains to those organizations. Such automated attacks, which change the behavior of applications without the explicit request of the end-users fall into a

category called Potentially Unwanted Programs (PUPs). PUPs are often bundled together with (seemingly) useful software, which leads end-users into unknowingly installing them. Once installed, PUPs change the behavior of popular programs by tampering with process memory, locally stored resources or the environment in which they run. Examples of PUP behavior include: changing the default search engine of a web-browser, aggressively displaying pop-up advertisements, tracking actions of end-users, causing an overall system slowdown and asking for fees to "fix performance". On the one hand, this change creates some form of financial gain for the organizations that own the PUPs. On the other hand, this change is detrimental for the vendor of the popular software and dangerous for its end-users. Recent work investigating the distribution of PUPs indicates that Google Safe Browsing generates on average over 60 million warnings related to PUPs per week, three times that of malware warnings [208]. Techniques employed by PUPs (e.g. code injection in the process memory, run-time memory patching, system call interposition) generally, do not raise any alarms in anti-virus software, because they are also performed by non-malicious third party software including anti-virus software, accessibility and graphics driver tools [207]. Some anti-virus products are able to detect PUPs. However, the vendors of popular software applications (e.g. web browsers) cannot assume that such anti-virus software is present on all end-user systems. Therefore, developers of popular applications incorporate protection mechanisms based on software diversity and obfuscation, inside of their own products, which introduce a tolerable amount of overhead and are transparent for end-users. Malicious MATE attackers therefore aim to develop PUPs which can bypass such protection mechanisms.

Benign MATE Not all software developers are benign. An example of malicious software developers are malware developers. Malware often performs illegal actions on the environment of a victim end-user, e.g. steal confidential information such as credit card numbers, passport numbers, passwords, etc. Malware developers also heavily employ obfuscation and software diversity because:

1. Diversely obfuscated binaries break signature-based malware detection, which lets malware developers infect as many end-users as possible.

2. Malware developers do not want any of these end-users (i.e. victims of their malware), to be able to reverse engineer the malware binaries and neutralize them.

Hence, benign MATE attackers are often malware analysts working for antivirus companies who want to understand what the malware is doing in order to disarm and remove it. Malware analysts are faced with millions of malware samples per day, which makes manual analysis unscalable. Therefore, they are forced to develop automated attacks which can handle diversely obfuscated code. These automated attacks will then be distributed as updates to the anti-virus engines of all end-users (probably having diverse instances of the malware), in order to stop the malware if it is present or detect it when it is transfered to the end-user's machine.

1.3. The Need for Characterizing the Strength of Software Obfuscation

In order to understand the some specific terms which will be used throughout the remainder of this thesis, this section describes the seminal work of Collberg et al. [58, 59], who proposed four dimensions for characterizing the quality of code transformations. These dimensions are:

- *Potency* against human-assisted analysis attacks.
- *Resilience* against automated analysis attacks.
- *Stealth* which refers to the effort of identifying the transformed (part of the) code inside a given program.
- *Cost* of the transformed program compared to the original program, which includes: run-time, memory and file size overhead.

Each of these dimensions can be associated with a discrete scale of values, e.g. low, medium and high. They can also be associated with one or more integer numbers. For instance, cost values are associated with metrics indicating the average or maximum overhead in terms of: run-time, memory usage and file sizes, for a certain set of program executions. However, for the values of potency, resilience and stealth it is not clear which measures to use. Collberg et al. [58] propose using various code complexity metrics for measuring potency, namely: program length [102], cyclomatic complexity [149], nesting complexity [103], data flow complexity [165], fan-in/-out complexity [105], data structure complexity [157], object oriented design metrics [47]. These metrics are believed to be correlated with the difficulty of understanding code for humans, nevertheless, there have been user studies which argued that this correlation is weak [199]. Collberg et al. [59] argue that the degree of stealth strongly depends on the program being transformed, because some transformations may produce stealthy code in some contexts (i.e. where the surrounding code is similar to the obfuscation output) and un-stealthy code in others. Several researchers have discussed possible measures of resilience [124, 7, 143, 155]. Despite the numerous efforts in this area, a recent survey of most of the common obfuscating transformations and deobfuscation attacks indicates that after more than two decades of research, we are still lacking reliable concepts for evaluating the resilience of code obfuscation against attacks [184].

Intuitively, a defender is chiefly interested in a quantifiable expression over his/her program and all attackers, saying that attacking a particular obfuscation transformation is bounded below by a certain work-factor. However, we believe this to be a very lofty goal. On the other hand, an attacker is mainly interested in developing an attack which outperforms any prior known attacks, especially if these attacks are not efficient in the attacker's context. We believe that these two perspectives complement each other. Therefore, we propose a model for quantifying obfuscation resilience by stipulating a – possibly hypothetical,

unknown, non-computable – lower bound. This reflects the perspective of the defender's interests. In practice, we are confined to providing *single data points*; their values define *upper bounds for the lower bounds*. These are interesting if a defender can conclude that even though s/he may not know the lower bound, available data already suggests that the obfuscation mechanism is too weak, assuming the *best known attackers*, according to today's knowledge.

1.4. Goal

The overarching goal of this thesis is to provide a framework for the quantitative characterization of the resilience of software code obfuscation transformations w.r.t. automated MATE attacks. This framework will aid the decision making process of an obfuscating party regarding which obfuscation transformations to employ for different scenarios involving automated attacks. Moreover, this framework will also guide the development of new obfuscation transformations to help defend against automate MATE attacks.

1.5. Problem Statement and Research Questions

Since there exist multiple obfuscation transformations and multiple automated MATE attacks, it is unclear how to quantify the effect of different obfuscation transformations w.r.t. different attacks. Moreover, it is unclear if cryptographic obfuscation may be of practical use for some scenarios, or which combination of practical code obfuscation techniques to choose when defending against a particular set of MATE attacks. Therefore, *the problem addressed in this work is that of characterizing the strength of obfuscated programs against automated MATE attacks*. To solve this problem statement and to achieve the goal of this thesis, the following research questions must be answered:

1. What is the overhead of using cryptographic obfuscation? (answered in Chapter 2)
2. What practical code obfuscation transformations have been proposed in the literature? (answered in Chapter 3)
3. How can we characterize the strength of obfuscation transformations using a general model that covers all attacks? (answered in Chapter 4)
4. Are there common (sub-)goals that must be achieved in order for a group of automated MATE attacks to be successful? (answered in Chapter 5)
5. Is there a way to determine an upper limit to the number of obfuscation transformations to apply? (answered in Chapter 5)
6. Which obfuscation transformations hinder automated attacks, by how much and at what cost? (answered in Chapter 5)

7. How can we determine which code features have the highest impact on different automated MATE attacks? (answered in Chapter 6)

8. How can we build obfuscation transformations that are stronger than current ones? (answered in Chapter 7)

9. What are the state of the art approaches for evaluating or characterizing the strength of obfuscation? (answered in Chapter 8)

1.6. Thesis Statement

This work focuses on answering the research questions from Section 1.5 in order to support the hypothesis that:

All automated MATE attacks involve search problems. The effort needed to solve such search problems can be quantified based on: (1) the attacker goal, (2) the characteristics of the program, which is the object of the attack, (3) the search strategy and (4) the heuristic function employed by attacker.

Elaborating on the hypothesis, the core of this work shows that by formulating automated MATE attacks as search problems, one is able to:

- Determine the program characteristics (e.g. size of program, McCabe cyclomatic complexity [149], etc.), which influence the effort of the automated MATE attack (see Chapter 4).

- Choose only those existing obfuscation transformations that affect program characteristics such that the automated MATE attack effort is increased (see Chapter 5).

- Predict the effort (e.g. time) needed to perform an automated MATE attack based on the characteristics of the program (see Chapter 6).

- Develop new obfuscation transformations that change the program characteristics such that the automated MATE attacks are prevented or hampered (see Chapter 7).

All of these previously enumerated results substantiate our thesis.

1.7. Solution

The general challenge of software obfuscation is to find transformations that are both practical and secure against any MATE attacker. In this respect we discuss how secure we can make practical obfuscations and how practical provably secure obfuscations are. However, we restrict the scope of this work to a special class of automated MATE attacks (i.e. attacks which use: symbolic execution, pattern matching, pattern recognition, taint analysis, etc.), because human assisted analysis is highly dependent on factors beyond our

control (e.g. knowledge, ingenuity, etc.) and relatively more expensive to perform, due to humans which need to be involved to perform attacks.

In this scope we propose a framework which is able to characterize the strength of obfuscation based on the effort of automated MATE attacks. Our framework is instantiated for common state-of-the-art obfuscation transformations and automated attacks based on symbolic execution. We implemented and published obfuscation transformations for which we did not find a freely available implementation. We then performed different case-studies where we obfuscated a set of programs and measured their impact on different attacks w.r.t. the original (unobfuscated) counterparts. The results lead to an intuition as to which obfuscation transformations can withstand which automated attacks given a certain amount of computing resources and time to execute the attack. More importantly, we identify key software features that are able to characterize the strength of obfuscation w.r.t. symbolic execution attacks. To show the importance of these features, we build a model leveraging them, in order to predict the effort needed by automated MATE attacks.

The benefits of our framework do not end with characterizing the strength of obfuscation based on the effort needed by automated MATE attacks. By using the information regarding which software features have a high impact on the attack effort, we are able to develop new obfuscation techniques that increase the effort by leveraging those features.

Since it is not possible to envision attack techniques that are yet to be published or developed, we do not claim that our results provide a lower bound on the resilience of an obfuscation transformation against arbitrary attacks. Instead, we provide an upper bound on the lower bound, i.e. we claim that the best attacker will not be worse than shown by our results. More importantly, we claim that the effort of any automated MATE attack – which is developed after the publication of this thesis – can be characterized using our framework. Therefore, the upper bound on the lower bound must be updated whenever a new attack becomes available.

1.8. Contributions

This thesis makes the following contributions:

- **A framework for characterizing the strength of obfuscation with respect to known automated MATE attacks.** The strength is measured using the effort needed by the best known attack. Using our framework we can formulate all automated MATE attacks as search problems, which facilitates reasoning about how to characterize the effort of the attack.

- **Several instantiations of our framework for various attacker goals and automated attacks.** We present an in-depth study of automated MATE attacks based on symbolic execution. We find that symbolic execution is able to bypass several popular obfuscation transformations with no human assistance. We discuss why different obfuscation transformations have different effects on the time needed to successfully complete a

symbolic execution attack. Moreover, we identify the most important features needed to characterize the strength of obfuscation against such attacks. We use these features to build a regression model which can predict the time needed for an attack, with high accuracy.

- **Implementations of obfuscation transformations that help improve resilience against specific automated attacks.** Based on our findings from the case-study on symbolic execution attacks, we develop two novel obfuscation transformations, which can exponentially increase the effort of such attacks.

Parts of the contributions of this thesis have previously appeared in the following peer-reviewed publications, co-authored by the author of this thesis:

1. **Banescu, S**; Collberg, C; Ganesh, V; Newsham, Z; Pretschner, A. *Code Obfuscation Against Symbolic Execution Attacks*. In Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC), 2016. **Best Paper Award**.

2. Salem, A; **Banescu, S**. *Metadata Recovery From Obfuscated Programs Using Machine Learning*. In Proceedings of the 6th Software Security, Protection and Reverse Engineering Workshop (SSPREW), 2016. **Best Paper Award**.

3. **Banescu, S**; Lucaci, C; Krämer, B; Pretschner, A. *VOT4CS: A Virtualization Obfuscation Tool for C#*. In Proceedings of 2nd International Workshop on Software Protection (SPRO), 2016.

4. **Banescu, S**; Wuechner, T; Salem, A; Guggenmos, M; Ochoa, M; Pretschner, A. *A Framework for Empirical Evaluation of Malware Detection Resilience Against Behaviour Obfuscation*. In Proceedings of 10th International Conference on Malicious and Unwandted Software (MALWARE), 2015.

5. **Banescu, S**; Ochoa, M; Pretschner, A. *A Framework for Measuring Software Resilience Against Automated Attacks*. In Proceedings of the 1st International Workshop on Software Protection (SPRO), 2015.

6. **Banescu, S**; Ochoa, M; Kunze, N; Pretschner, A. *Idea: Benchmarking indistinguishability obfuscation - A candidate implementation*. In Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS), 2015.

In addition to the previously enumerated papers, the author of this thesis has co-authored the following peer-reviewed publications, which tackle relevant problems, related to the topic of this thesis, but are not part of this thesis:

8. **Banescu, S**; Ahmadvand, M; Pretschner, A; Shield, R; Hamilton, C. *Detecting Patching of Executables without System Calls*. In Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY), 2017.

9. Ibrahim, A; **Banescu, S**. *StIns4CS: A State Inspection Tool for C#*. In Proceedings of 2nd International Workshop on Software Protection (SPRO), 2016.

10. Fedler, R; **Banescu, S**; Pretschner, A. *ISA2R: Improving Software Attack and Analysis Resilience via Compiler-Level Software Diversity*. In Proceedings of 34th International Conference on Safety, Reliability, and Security (SAFECOMP), 2015.

11. **Banescu, S**; Pretschner, A; Battre, D; Cazzulani, S; Shield, R; Thompson, G. *Software-Based Protection against "Changeware"*. In Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY), 2015.

1.9. Structure

Chapter 2 provides an overview of theoretical obfuscation constructions and issues that hinder their application in practice. Chapter 3 provides a classification and a survey of obfuscation transformations. Chapter 4 describes our framework for characterizing the strength of obfuscation against automated MATE attacks. Chapter 5 presents a case study where we instantiate the framework for dynamic symbolic execution attacks with the goal of extracting a license key from obfuscated programs. Chapter 6 presents another case study where we aim to extract the most relevant features that characterize the effort of a symbolic execution attack, in order to build prediction models to estimate the time needed for such attacks. Chapter 7 presents novel obfuscation techniques to hinder symbolic execution attacks. Chapter 8 presents related work. Chapter 9 presents conclusions, insights and future work.

2. Obfuscation in Theory

This chapter presents a brief overview of cryptographic obfuscation and why it is currently far from being practical. Parts of this chapter have previously appeared in a peer-reviewed publication [14], co-authored by the author of this thesis.

The first formal study of obfuscation was published in 2001 by Barak et al. [26]. They proposed that an ideal obfuscator should be able to take any program and transform it into a *virtual black box*, i.e. a MATE attacker would be able to interact with it in the same manner as with a program running on a remote server, however, the attacker would not be able to learn anything from the program in addition to what can be learned from its input-output behavior. In Section 2.1 we show the formal definition of this ideal (black-box) obfuscator as given in [26], as well as a sketch of the proof that such an obfuscator cannot exist.

Over a decade later, Garg et al. [90] proposed a construction for *indistinguishability obfuscation*, a different obfuscation notion than black-box obfuscation, which guarantees that the obfuscations of two programs implementing the same functionality are computationally indistinguishable. This was a major breakthrough in cryptography, since a few years earlier it was proven by Goldwasser and Rothblum [97] that indistinguishability obfuscation is the best possible type of obfuscation that can be achieved for all programs. Therefore, we are currently seeing a revival of interest in obfuscation from the cryptographic community, because the construction of Garg et al. [90] may be employed to construct functional encryption, public key encryption, digital signatures, etc. We describe this construction in Section 2.2. Afterwards, we present our own implementation of this construction and its applicability in practice in Section 2.3

2.1. Impossibility of Black-Box Obfuscation

As opposed to practical obfuscation which describes transformations directly on computer programs, cryptographic obfuscation often talks about transformations on *boolean circuits*, which can be translated to computer programs, however, they are not as expressive as most programing languages used in practice (e.g. boolean circuits do not allow loops such as C/C++, Java, etc.). A boolean circuit C is a directed acyclic graph, where nodes are represented by conjunction, disjunction and/or negation gates with maximum 2 inputs (fan-in-2), which process only boolean values. The *inputs* of a circuit are all gates with in-degree 0, while the *outputs* are gates with out-degree 0. If C has n inputs and $x \in \{0,1\}^n$, then we denote by $C(x) \in \{0,1\}^m$, the m-bit output of C when given input x. Therefore, a

circuit C can be defined as a function $C : \{0,1\}^n \to \{0,1\}^m$. The *size* of a circuit (denoted $|C|$) is equal to the total number of gates in that circuit. The following sections will also overload the semantics of the vertical bars ($|\cdot|$) operator, which denotes the absolute value when applied to a real number. The *depth* of a circuit is the length of the longest path from input to output gate, in the circuit. If S is a Probabilistic Polynomial Time Turing Machine (PPT), we denote by $S^C(x)$ the output of S when given input x and oracle access to the circuit C. Oracle access is not limited to a single circuit, e.g, $S^{C,D}(x)$ denotes the output of S when given input x and oracle access to both circuits: C and D. Finally, note that circuits are also represented using a string of binary digits of a certain maximum size. Therefore, circuits can also be treated as input data of PPTs.

2.1.1. Definition of Black-Box Obfuscation

In this context an obfuscator must satisfy three properties. Firstly it must preserve the input-output behavior of its input program. Secondly, it must not induce more than a polynomial overhead. Finally, a PPT attacker must not able to compute any predicate (property) of the original program from the obfuscated program, which property the attacker could not compute given only oracle access to that original program. These properties are formally described in the following definition.

Definition 2.1. *(Circuit Obfuscator [26]) A probabilistic algorithm $\mathcal{O}$ is a circuit obfuscator if the following conditions hold:*

- *(functionality) For every circuit C, the string $\mathcal{O}(C)$ describes a circuit that computes the same function as C.*
- *(polynomial slowdown) The description length and running time of $\mathcal{O}(C)$ are at most polynomially larger than that of C. That is, there is a polynomial p such that for every circuit C, $|\mathcal{O}(C)| \leq p(|C|)$.*
- *(virtual black box property) For any PPT A, there is a PPT S and a negligible function α such that for all circuits C*

$$\left|\Pr[A(\mathcal{O}(C)) = 1] - \Pr[S^C(1^{|C|}) = 1]\right| \leq \alpha(|C|).$$

We say that $\mathcal{O}$ is efficient if it runs in polynomial time.

2.1.2. Sketch of Impossibility Proof

Barak et al. [26] state in a theorem that such an obfuscator does not exist for *all* circuits (programs). However, note that the theorem does not say that it is impossible to build a black-box obfuscator for a *particular set* of circuits (programs). Moreover, the virtual black-box property is not always necessary in practice. Therefore, this impossibility result has not discouraged other researchers looking into practical obfuscation transformations.

The proof of the previous theorem is based on a counter example. In this section we do not provide the entire proof, instead, we only show the counter example to give the reader an intuitive understanding of the impossibility result and we refer to the original paper [26] for the full proof.

If a black-box obfuscator exists for all programs, then even if an attacker is given two or more obfuscated programs, s/he should not be able to infer any property about any of these programs by combining them in some way. This statement is defined by Barak et al. [26] as:

Definition 2.2. *(2-circuit Obfuscator [26]) A 2-circuit obfuscator is defined in the same way as a circuit obfuscator (see Definition 2.1), except that the "virtual black-box" property is replaced by the following:*

- *(virtual black-box property) For any PPT A, there is a PPT S and a negligible function α such that for all circuits C, D*

$$\left|\Pr[A(\mathcal{O}(C),\mathcal{O}(D))=1]-\Pr[S^{C,D}(1^{|C|+|D|})=1]\right|\leq\alpha(\min\{|C|,|D|\})$$

Barak et al. [26] show that 2 circuits can be merged into one. Hence, after proving that 2-circuit obfuscators do not exist for all programs, it is straightforward to prove that circuit obfuscators do not exist as well. Here we will only show the proof that 2-circuit obfuscators do not exist for all programs. The essence of the proof is that there is a fundamental difference between having oracle access to a function and having a program that computes that function. If a function is (exactly) *learnable* via a polynomial number $p(k)$ of queries to the oracle, then this difference is insignificant. Therefore, the proof assumes the existence of *one-way functions* which are *unlearnable* by a PPT attacker via queries to an oracle, e.g.:

$$C_{\alpha,\beta}(x)=\begin{cases}\beta & \text{if } x=\alpha\\ 0 & \text{otherwise}\end{cases}$$

The second circuit $D_{\alpha,\beta}$ that will be obfuscated is a *point function*, which interprets its input as a function (C), and distinguishes whether this function outputs a particular value (β) when given a particular input (α), i.e.:

$$D_{\alpha,\beta}(C)=\begin{cases}1 & C(\alpha)=\beta\\ 0 & \text{otherwise}\end{cases}$$

The MATE attacker is a PPT A, which given two circuits as arguments, simply applies the second argument on the first argument, i.e. $A(C,D)=D(C)$. Hence, if $C_{\alpha,\beta}$ and $D_{\alpha,\beta}$ can be represented with $\Theta(k)$ bits of information, then for any $\alpha,\beta\in\{0,1\}^k$,

$$\Pr[A(\mathcal{O}(C_{\alpha,\beta},\mathcal{O}(D_{\alpha,\beta})=1]=1 \tag{2.1}$$

On the other hand, a PPT attacker S, with only oracle access to $C_{\alpha,\beta}$ and $D_{\alpha,\beta}$ will only have a probability of $2^{-\Omega(k)}$ of guessing an input that will cause either of the two oracles to produce an output different from 0, i.e.:

$$\left|\Pr[S^{C_{\alpha,\beta},D_{\alpha,\beta}}(1^k)=1]-\Pr[S^{Z_k,D_{\alpha,\beta}}(1^k)=1]\right|\leq 2^{-\Omega(k)}, \tag{2.2}$$

where Z_k is a circuit which outputs zero for all inputs. However, from the definition of the MATE attacker A we have:

$$\Pr[A(\mathcal{O}(Z_k),\mathcal{O}(D_{\alpha,\beta}))=1]=0 \tag{2.3}$$

Equations 2.1, 2.2 and 2.3 show that there does not exist a 2-circuit obfuscator for all programs, because there exists a class of functions for which the virtual black-box property does not hold.

Note that this proof has focused on an adversary which aims to compute any 1-bit predicate (property) of a program. In Barak et al. [26], we also find a more general type of adversary, than the one who wants to compute a property of the program. This adversary's goal is to generate an output distribution given only oracle access to P, which is computationally distinguishable from anything s/he can compute given $\mathcal{O}(p)$. This type of adversary is the focus of indistinguishability obfuscation, which we discuss in Section 2.2.

2.2. Indistinguishability Obfuscation

This section presents the definition and candidate indistinguishability obfuscation construction developed by Garg *et al.* [90] applied to *boolean circuits* in $\mathbf{NC}^1$ [8], preceded by the concepts needed to understand this construction.

An *indistinguishability obfuscator* must satisfy two properties: (1) it must preserve the input-output behavior of the unobfuscated circuit and (2) given two circuits $C_1, C_2 \in \mathcal{C}_\lambda$ and their obfuscated counterparts $i\mathcal{O}(\lambda, C_1), i\mathcal{O}(\lambda, C_2)$, a PPT adversary will not be able to distinguish which obfuscated circuit originates from which original circuit with significant probability (the advantage of the adversary is bounded by a negligible function of the security parameter λ). This definition is formally specified by Garg et al. [90] in the following definition.

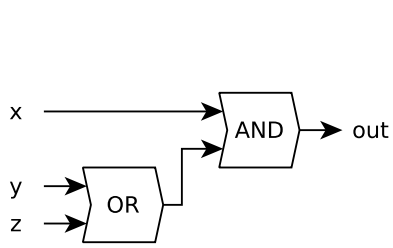
Definition 2.3. *(Indistinguishability Obfuscator ($i\mathcal{O}$) [90]) A uniform PPT $i\mathcal{O}$ is called an* indistinguishability obfuscator *for a circuit class $\{\mathcal{C}_\lambda\}$ if the following conditions are satisfied:*

- *For all security parameters $\lambda \in \mathbb{N}$, for all $C \in \mathcal{C}_\lambda$, for all inputs x, we have that*

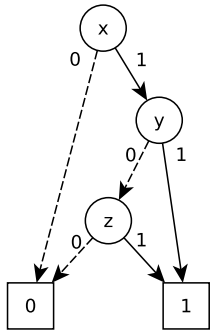
$$Pr[i\mathcal{O}(\lambda, C)(x) = C(x)] = 1$$

- *For any PPT distinguisher D, there exists a negligible function α such that the following holds: For all security parameters $\lambda \in \mathbb{N}$, for all pairs of circuits $C_0, C_1 \in \mathcal{C}_\lambda$, we have that if $C_0(x) = C_1(x)$ for all inputs x, then*

$$|Pr[D(i\mathcal{O}(\lambda, C_0)) = 1] - Pr[D(i\mathcal{O}(\lambda, C_1)) = 1]| \leq \alpha(\lambda)$$



(a) Boolean circuit.



(b) Branching Program (BP).

Figure 2.1.: Boolean circuit and its corresponding Branching Program (BP).

The previous definition does not indicate how to construct $i\mathcal{O}$. In the following we present the mathematical building blocks needed for the candidate construction proposed by Garg et al. [90].

2.2.1. Branching Programs

Even though at an abstract level $i\mathcal{O}$ applies to boolean circuits, in the candidate construction proposed by Garg et al. [90] all circuits are transformed into Branching Programs (BPs). A BP – also called a Binary Decision Diagram (BDD) – is a rooted, connected, directed, acyclic graph, which is used to compute a boolean function. A BP has two types of nodes, namely decision nodes with out-degree equal to two and terminal nodes with out-degree equal to zero. All decision nodes are associated with exactly one input variable – representing an input of the circuit – and the two outgoing arcs represent logical values 0 and 1 assigned to the input associated to that node. All terminal nodes are of two types, namely logical 0 or 1, representing the output value of the circuit. Figure 2.1b shows an example of a BP corresponding to the simple circuit from Figure 2.1a, which computes the following boolean function: $f(x, y, z) = x \wedge (y \vee z)$. A *layer* of a BP is defined as the set of non-terminal nodes with the same distance from the root node. All nodes in the same layer are associated with the same input variable. The BP in Figure 2.1b has three layers corresponding to input variables x, y and z. The *length* of the BP is equal to its number of layers.

In [90], each layer i of a BP is encoded as two square permutation matrices of size m, i.e. $A_{i,0}, A_{i,1} \in \{0,1\}^{m \times m}$. The two permutation matrices correspond to the logical values 0 and 1, that may be assigned to the input variable associated to layer i. The result of such a BP is computed by choosing one of the two permutation matrices corresponding to the logical value assigned to the input variable associated to each layer. These matrices are then multiplied and the result is compared with two pre-computed permutation matrices, corresponding to the two terminal nodes 0 and 1, i.e. $A_0, A_1 \in \{0,1\}^{m \times m}$. This encoding of a BP is called an *oblivious linear branching program* and is formally defined subsequently.

Definition 2.4. *(Oblivious Linear Branching Program [90]) Let $A_0, A_1 \in \{0,1\}^{m\times m}$ be two distinct arbitrarily chosen permutation matrices. An (A_0, A_1) oblivious BP of length n for circuits with ℓ-bit inputs, is a sequence of instructions $BP = ((inp(i), A_{i,0}, A_{i,1}))_{i=1}^{n}$, where $A_{i,b} \in \{0,1\}^{m\times m}$, and $inp : \{1,n\} \to \{1,\ell\}$ is a mapping from BP instruction index to circuit input bit index. The function computed by the BP is*

$$f_{BP,A_0,A_1}(x) = \begin{cases} 0 & \text{if } \Pi_{i=1}^{n} A_{i,x_{inp(i)}} = A_0 \\ 1 & \text{if } \Pi_{i=1}^{n} A_{i,x_{inp(i)}} = A_1 \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

The transformation from a circuit to an oblivious linear branching program (hereafter simply BP) is made possible by Barrington's theorem [29], which states that any fan-in-2, depth-d boolean circuit (i.e. all circuits from class $\mathbf{NC}^1$ [8]) can be transformed into a BP of length at most 4^d using only permutation matrices of size 5×5, that computes the same function as the circuit.

2.2.2. Universal Circuits and Kilian's Protocol

The family of circuits $\mathcal{C}_\lambda$ is characterized by ℓ inputs, λ gates, $O(log\lambda)$ depth and one output. $\mathcal{C}_\lambda$ has a corresponding polynomial-sized Universal Circuit (UC), which is a function $U_\lambda : \{0,1\}^{f(\lambda)} \times \{0,1\}^{\ell} \to \{0,1\}$, where $f(\lambda)$ is some function of λ. U_λ can encode all circuits in $\mathcal{C}_\lambda$, i.e. $\forall C \in \mathcal{C}_\lambda, \forall z \in \{0,1\}^{\ell}, \exists C_b \in \{0,1\}^{f(\lambda)} : U_\lambda(C_b, z) = C(z)$. It is important to note that the input of U_λ is a $f(\lambda) + \ell$ bit string and that by fixing any $f(\lambda)$ bits, one obtains a circuit in $\mathcal{C}_\lambda$.

UCs are part of the candidate $i\mathcal{O}$ construction, because they enable running Kilian's protocol [128], which allows two parties (V and E), to evaluate any $\mathbf{NC}^1$ circuit (e.g. U_λ) on their joint input $\mathcal{X} = (x|y)$, without disclosing their inputs to each other, where x, y are the inputs of V, respectively E. This is achieved by transforming the circuit into a BP, $BP = ((inp(i), A_{i,0}, A_{i,1}))_{i=1}^{n}$ by applying Barrington's theorem [29]. Subsequently, V chooses n random invertible matrices $\{R_i\}_{i=1}^{n}$ over $\mathcal{Z}_p$, computes their inverses and creates a new Randomized Branching Program (RBP), $RBP = ((inp(i), \tilde{A}_{i,0}, \tilde{A}_{i,1}))_{i=1}^{n}$, where $\tilde{A}_{i,b} = R_{i-1}A_{i,b}R_i^{-1}$ for all $i \in \{1,n\}, b \in \{0,1\}$ and $R_0 = R_n$. It can be shown that RBP and BP compute the same function. Subsequently, V sends E only the matrices corresponding to her part of the input $\{\tilde{A}_{i,b} : i \in \{1,n\}, inp(i) < |x|\}$ and E only gets the matrices corresponding to one specific input via oblivious transfer. E can now compute the result of RBP without finding out V's input. Kilian's protocol is related to the notion of program obfuscation, if we think of V as a software vendor who wants to hide (obfuscate) a program that is going to be distributed to end-users (E). However, Kilian's protocol [128] is modified in [90], by sending all matrices corresponding to any input of E, which allows E to run the RBP with more that one input. This modified version is vulnerable to *partial evaluation attacks*, *mixed input attacks* and also non-multilinear attacks, which extract information about the secret input of V.

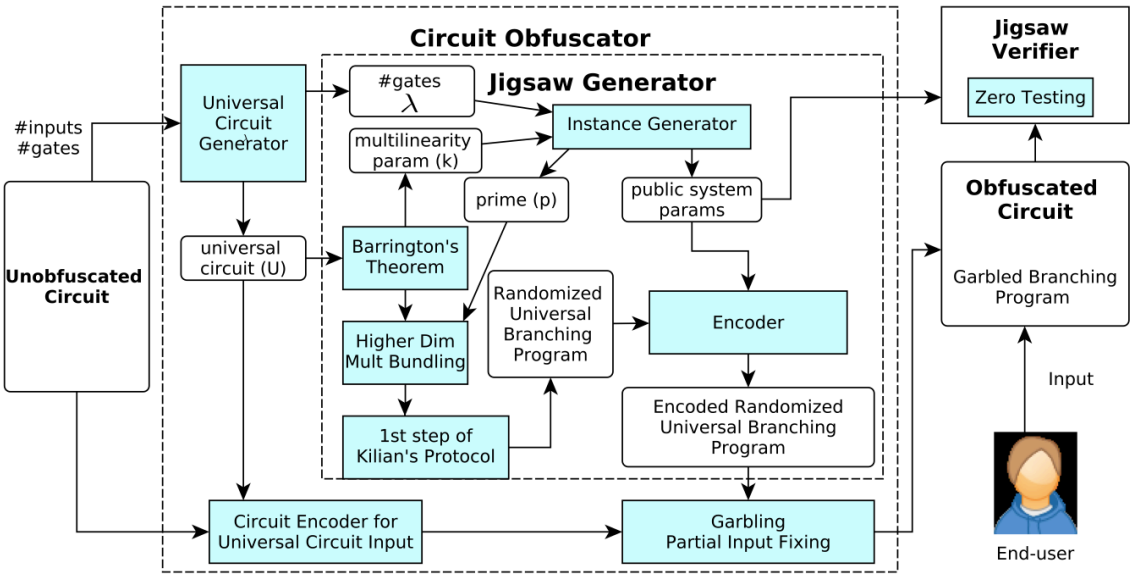


Figure 2.2.: Overview of the candidate construction for indistinguishability obfuscation

To prevent partial evaluation attacks Garg *et al.* [90] transform the 5×5 matrices of BP into higher order matrices, having dimension $2m + 5$, where $m = 2n + 5$ and n is the length of BP. Subsequently, they add 2 *bookend* vectors of size $2m + 5$ in order to neutralize the multiplication with the random entries in the higher order matrices. To prevent mixed input attacks a multiplicative bundling technique is used, which leads to an *encoded* output of BP. To decode the output of the BP an additional BP of equal length with BP, which computes the constant 1 function, is generated and the same multiplicative bundling technique is applied to it. Subtracting the results of the two BPs executed on the same inputs, will decode the output of BP. To prevent non-multilinear attacks, the candidate construction of Garg *et al.* [90] employs the Multi-linear Jigsaw Puzzle (MJP).

2.2.3. Multilinear Jigsaw Puzzle (MJP)

An overview of MJP is illustrated in Figure 2.2 and consists of two entities, i.e. the *Jigsaw Generator* (JGen) and the *Jigsaw Verifier* (JVer). The JGen is part of the *circuit obfuscator*. It takes as input a security parameter (λ), a UC (U_λ) and the number of input bits (ℓ) of any circuit simulated by U_λ. JGen first applies Barrington's theorem [29] to transform U_λ into a Universal Branching Program (UBP), UBP of length n. Subsequently, the *Instance Generator* takes λ and the multilinearity parameter ($k = n + 2$) as inputs and outputs a prime number p and a set of public system parameters (including a large random prime q and a small random polynomial $g \in \mathcal{Z}[X]/(X^m + 1)$). Afterwards, UBP is transformed into a RBP by: (1) transforming the BP matrices into higher order matrices, (2) applying multiplicative

bundling and (3) the first step of Kilian's protocol. The output of JGen is a set of public system parameters and the randomized UBP ($\widehat{\mathcal{RND}}(UBP_\lambda)$) with all matrices encoded by the *Encoder* component.

The output of JGen can be used to obfuscate a circuit $C \in \mathcal{C}_\lambda$ by fixing a part of the inputs (garbling) of $\widehat{\mathcal{RND}}(UBP_\lambda)$ such that it encodes C for all $z \in \{0,1\}^\ell$. Garbling is done by discarding the matrices of $\widehat{\mathcal{RND}}(UBP_\lambda)$ which correspond to values not chosen for the fixed input bits. The result of this step is $i\mathcal{O}(\lambda, C)$, the candidate of Garg *et al.* [90]. It is sent to an untrusted party which evaluates it by fixing the rest of its inputs and providing it as input to the JVer. The JVer outputs 1 if the evaluation of $i\mathcal{O}(\lambda, C)$ is successful and 0, otherwise.

2.3. Applicability in Practical Scenarios

Although the proposers of indistinguishability obfuscation acknowledge that their construction is not practical as of today [75], concrete details had not been published at the time that the author of this thesis co-authored [14]. The motivation of [14] was thus to better understand how far the candidate construction is from being used in real applications. To do so, we prototypically implemented the algorithm described in [90] and benchmarked its space and time performance depending on various parameters. Details are presented in the following.

2.3.1. Implementation

Our proof-of-concept implementation was done in Python, leveraging the SAGE computer algebra system and can be downloaded from the Internet [93]. It consists of the following modules, corresponding to the light blue rectangles from Figure 2.2: (1) building blocks for UC creation, (2) Barrington's theorem for transforming boolean circuits to BPs, (3) transformation from BP matrices into higher order matrices and applying multiplicative bundling (4) 1st step of Kilian's protocol for creating RBPs from BPs, (5) instance generator for MJP, (6) encoder for MJP, (7) circuit encoder into input for UC, (8) partial input fixer for RBPs, and (9) zero testing of jigsaw verifier.

Technical challenges faced Although commonly used in the literature, we could not find a readily available implementation of Universal Circuits (UC) that was easily adaptable to our setting. Therefore we decided to implement our own UC component, following the less performant algorithm of [182]. For the sake of performance, this component can be improved by following for instance the more performant (but more complex) algorithm suggested in [182] or [210].

Challenges interpreting [90] We also faced some challenges while interpreting the candidate construction description, in particular their suggested encoding function. For instance it was difficult to come up with concrete values for some parameters, since the relation

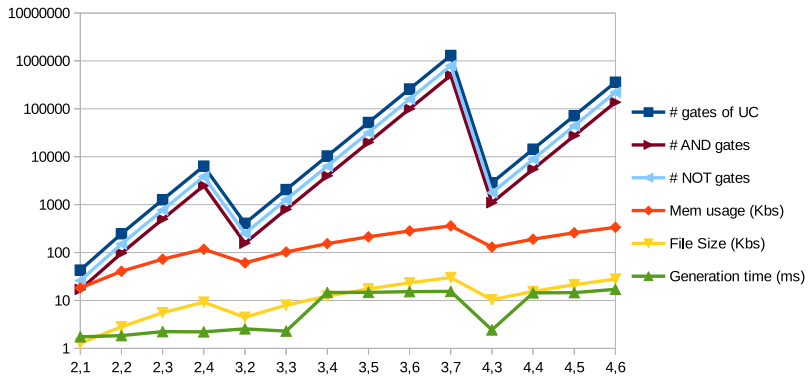


Figure 2.3.: Generation of UCs (X-axis: no. inputs (ℓ), no. gates of input circuit (λ))

between them is given using the big O notation. On the other hand, the Encoder function requires to reduce an element $a \in \mathcal{Z}_p$ modulo a polynomial g of degree ≥ 1. We could not think of a better canonical representative for this reduction than a itself, which makes us believe that either the modulo reduction is redundant or Garg et al. [90] had another canonical representative in mind (a polynomial) which is unclear how to compute.

Summary of current status Currently, our implementation can perform most steps of the candidate construction, with the exception of the zero test. We believe this is a result of an incorrect choice of the canonical representative of a modulo g or/and of the concrete parameters as discussed above. We have raised these issues in popular mathematics and cryptography forums and contacted Garg et al. [90] for clarification with no success at the moment of elaborating this document. However, note from Figure 2.2 that the improper functioning of the zero test does not affect the results of benchmarking the *Circuit Obfuscator* presented in the next section, because it is part of the *Jigsaw Verifier*.

2.3.2. Benchmarking

We executed our experiments on a machine with average hardware, i.e. four 2,6 GHz cores and 64 GBs of memory. The first experiment aims to investigate the resources required to obfuscate a circuit consisting only of AND gates as a function of its number of inputs and gates. As illustrated in Figure 2.2 the first step of obfuscation consists of generating the UC, corresponding to the first step of our experiment. The number of circuit inputs were varied between 2 and 4, while the number of gates between 1 and 10. The recorded outputs are shown in Figure 2.3 and consist of the: number of gates, memory usage, output file and generation time needed for the UC. Observe that increasing the number of inputs causes a linear increase in each measured output of the experiment, while increasing the number of gates causes an exponential increase. The memory usage is around one order of magnitude higher than the file size due to the compression algorithm we use to store UCs.

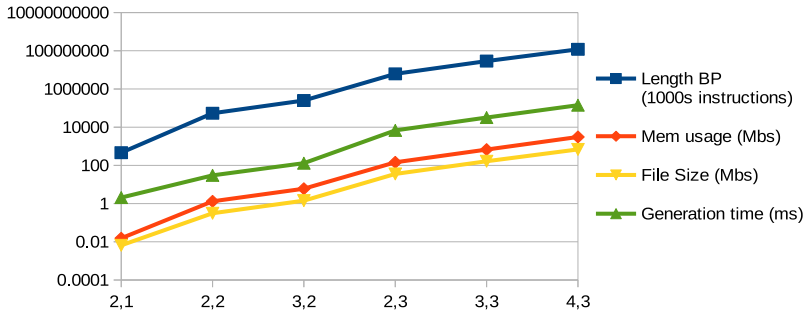


Figure 2.4.: Generation of BPs (X-axis: no. inputs (ℓ), no. gates of input circuit (λ))

The second step of our experiment consisted of transforming the previously generated UCs into BPs using our implementation of Barrington's theorem [29]. However, it was infeasible to transform all the previously generated UCs because of the fast polynomial increase in memory usage and file size, illustrated in Figure 2.4. We estimated the size of generating a BP for a UC which encodes a circuit by applying following recursive formula (corresponding to our implementation), to the output gate of a UC:

$$l(\mathsf{gate}) = \begin{cases} 1 & \text{if } \mathsf{type}(\mathsf{gate}) = \mathsf{Input} \\ l(\mathsf{gate.input}) & \text{if } \mathsf{type}(\mathsf{gate}) = \mathsf{NOT} \\ 2l(\mathsf{gate.input1}) + 2l(\mathsf{gate.input2}) & \text{if } \mathsf{type}(\mathsf{gate}) = \mathsf{AND} \end{cases}$$

The estimated memory usage of a universal BP which encodes 4 inputs and 6 gates, corresponding to the largest UC we show in Figure 2.3, is over 4.47 Peta Bytes, which is infeasible to generate on our machine.

The third step of our experiment was to transform the BPs generated previously into RBPs by transforming the BP matrices into higher order matrices, applying multiplicative bundling and the first step of Kilian's protocol [128]. The results of this experiment are shown in Figure 2.5. Additionally to the number of inputs and gates, in this experiment we also have the matrix dimension increase (m) and the choice of the prime (p) corresponding to $\mathcal{Z}_p$ in which Kilian's protocol operates. The choice of m influences both the generation time and the file size polynomially. Observe that the memory usage remains constant for different values of m. This is due to compatibility issues between SAGE and our memory profiler. However, we observed that the actual memory usage is still one order of magnitude higher than the file size. On the one hand, p influences the generation time linearly. On the other hand, the memory usage and file size are affected only if the data type width (range) of p grows, e.g. from a byte to an integer. Note that, the memory usage is not shown in Figure 2.5 since it could not be measured reliably due to technical limitations of our memory profiler. We estimate that the memory usage is approximately one order of magnitude higher than the file size.

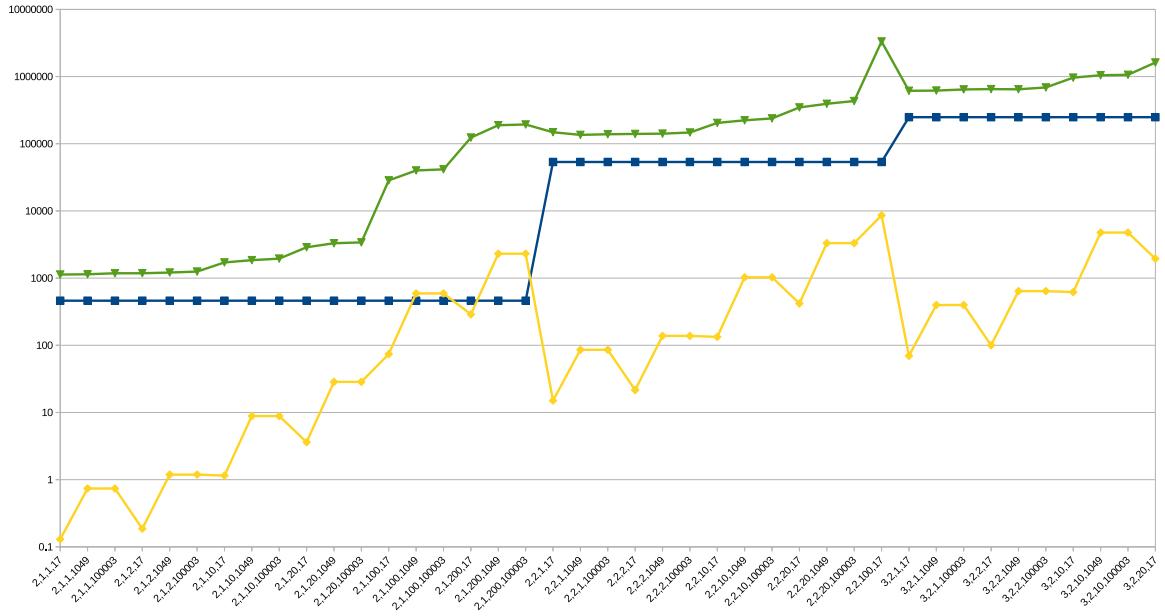


Figure 2.5.: Generation of RBPs (X-axis: no. inputs (ℓ), no. gates of input circuit (λ), matrix dimension (m), prime number (p)). Legend is the same as Figure 2.4.

2.4. Summary

In this section we have presented a non-trivial upper bound on the size and performance of the obfuscated versions of small circuits. To give an idea about the practicality of this construction, consider a 2-bit multiplication circuit. It requires 4 inputs and between 1 and 8 AND gates for each of its 4 output bits. An obfuscation would be generated in about 10^{27} years on a 2,6 GHz CPU and would require 20 Zetta Bytes of memory for $m = 1$ and $p = 1049$. Executing this circuit on the same CPU would take 1.3×10^8 years. This clearly indicates that for the time being the candidate construction is highly unpractical.

However, this upper bound can still be tightened (perhaps even dramatically) by improving upon our preliminary implementation. In particular, there exist better algorithms for the generation of UCs, which directly affect the size of the obfuscation [182, 210]. There is an inherent limitation for this improvement due to the fact that the output of gates in UCs are reused by other gates, which causes duplication of matrices in BPs when using Barrington's theorem [29]. Therefore, one improvement is to avoid using Barrington's theorem as suggested by Ananth et al. [4]. On the other hand, we have only implemented the construction for $\mathbf{NC}^1$ circuits: the candidate construction includes an extension to cope with bigger circuit classes, that includes the use of fully homomorphic encryption. As research advances towards practical fully homomorphic encryption, we expect our open implementation of the candidate indistinguishability obfuscation algorithm to foster improvements by the community.

3. Obfuscation in Practice

This chapter presents an overview of obfuscation and software diversity transformations employed in practice. It also describes challenges of building practical obfuscators. Parts of this chapter have also appeared in a publication [22], co-authored by the author of this thesis.

A practical obfuscator is in essence a compiler that takes a program as input, and outputs a functionally equivalent program, which is harder to understand and analyze than the input program. The meaning of the phrases "functionally equivalent" and "harder to understand and analyze" are different in practice than in theory (see Chapter 2) and this difference will be discussed in this chapter. For instance, some classical compiler optimizations are also considered obfuscation transformations, because in order to make the code more efficient, such optimizations may replace control-flow abstractions that are easy to understand by developers (e.g. loops), with other constructs which are less straightforward (e.g. goto statements).

One of the first works which proposed practical code (obfuscation) transformations for the purpose of software diversity, was published by Cohen in 1993 [53]. In his work, Cohen describes the *ultimate attack* (which is equivalent to the MATE attacker we described in Chapter 1), and the *ultimate defense* in this context, which can be done by increasing the complexity of the attack by means of code (obfuscation) transformations, trusted hardware or a combination of the two. Cohen also proposes a set of thirteen code transformations, which he suggests should be mixed according to the application and the security goals of the software developer. The work of Cohen has been extended by numerous authors. Some authors have focused on obfuscation [58, 184], others on software diversity [134, 31], however, it is clear that by employing (different) obfuscation transformations on the same input program we can obtain a multitude of diverse software instances.

3.1. Practical Challenges of Code Transformations

This chapter provides a conceptual overview of code transformations and does not focus on any particular implementation. Nevertheless, it is important to also keep implementation challenges in mind when talking about code transformations. Therefore, here we give an indication of one of the most important challenges facing obfuscation implementations in practice, i.e. correctness.

As stated in the informal definition of obfuscation, at the beginning of this chapter: obfuscators are a type of compilers. Therefore, similarly to compilers, obfuscators may introduce bugs, i.e. change the IO behavior (functionality) of their input program [106]. Since such a change in functionality is undesirable, several researchers have tackled this problem in the field of compilers. One approach for solving this issue is to build a formally verified compiler [136]. Nonetheless, such a task is, for the time being, time consuming and resource intensive for practical compiler developers.

Another approach for solving the previous issue is to directly test if the output of the compiler (obfuscator) is functionally equivalent to its input. However, from Rice's theorem we know that this problem is in general undecidable. Nevertheless, there are works [118, 192, 82], which employ equivalence checking heuristics on logic circuits or programs that can be clearly mapped onto logic circuits. Conversely, Holling et al. [109] have proposed an automatic tool for testing non-equivalence of programs, i.e. given a certain time budget the tool performs a symbolic analysis on the input and output programs and tries to find inputs for which outputs do not match. This is less expensive to perform than equivalence checking, nonetheless, it has issues with scalability as well.

The most scalable and successful approach for solving the previous issue is to directly test compilers for bugs. Random testing has been applied to many compilers for over 50 years with relatively high levels of success [34]. Yang et al. [229] employed random differential testing [151], i.e. randomly generating C programs and comparing the outputs of several compilers, e.g. different versions of the GNU C Compiler (GCC) and Low Level Virtual Machine (LLVM) compilers. They found more than 300 previously unknown bugs.

3.2. Classification of Code Obfuscation and Diversity Transformations

Several surveys and taxonomies for software obfuscation and diversity have been proposed in the literature [58, 13, 147, 134, 31, 184]. This section describes the most common classification dimensions presented in those works. The following sections present the actual transformations based on these dimensions.

3.2.1. Abstraction Level of Transformations

One common dimension of code transformations is the level of abstraction at which these transformations have a noticeable effect, i.e. *source code*, *intermediate representation* and *binary machine code*. Such a distinction is relevant for usability purposes, e.g. a JavaScript developer will mostly be interested in source code level transformations and a C developer will mainly be interested in binary level. However, none of the previously mentioned taxonomies and surveys classify transformations according to the abstraction level. This is due to the fact that some obfuscation transformations have an effect at multiple abstraction levels.

Moreover, it is common for papers to focus only on a specific abstraction level, disregarding transformations at other levels.

3.2.2. Time of Transformations

The time point at which a transformation can be applied is a classification dimension proposed by Larsen et al. [134]. The possible times when a transformation can be employed are: *implementation*, *compilation & linking*, *installation*, *loading*, *execution* and *update*. This dimension is related to the abstraction level of transformations, from Section 3.2.1, because implementation time is always associated with the source code level of abstraction and compilation & linking is always associated with intermediate representation. However, it is not completely overlapping, e.g. Java bytecode is an intermediate representation which can be transformed also after compilation & linking. Applying code transformations earlier in the development or distribution stage may or may not provide a higher level of security [140]. Moreover, it may also be more costly for the developer to do so [53, 87, 23].

3.2.3. Unit of Transformations

Larsen et al. [134] proposed classifying transformations according to the of granularity at which they are applied. Therefore they propose the following levels of granularity:

- *Instruction level* transformations are applied to individual instructions or sequences of instructions. Larsen et al. [134] assume the intermediate representation level of abstraction. However, this unit of transformation can be translated easily to binary machine code. Also at source code level we can consider a code statement as one or more instructions.

- *Basic block level* transformations affect the position of one or more basic blocks. Basic blocks are a list of sequential instructions that have a single entry point and end in a branch instruction.

- *Loop level* transformations alter the familiar loop constructs added by developers.

- *Function level* transformations affect several instructions and basic blocks of a particular subroutine. Moreover, they may also affect the stack and heap memory corresponding to the function.

- *Program level* transformations affect several functions inside an application. However, they also affect the data segments of the program and the memory allocated by that program.

- *System level* transformations target the operating system or the runtime environment and they affect how other programs interact with them.

The unit of transformation is important in practice because developers can choose the appropriate level of granularity according to the asset they must protect. For example, loop level transformations are not appropriate for hiding data, but they are appropriate for hiding algorithms. However, the same problem, as for the previous classification dimensions, arises for the unit of transformation, namely the same obfuscation transformation may be applicable to different units of transformation.

3.2.4. Dynamics of Transformations

Another classification dimension which is related to the time of transformation from Section 3.2.2, is the dynamics of transformation used by Schrittwieser et al. [184]. The dynamics of transformation indicate whether a transformation is applied to the program or its data *statically* or *dynamically*. Static transformations are applied once during: implementation, compilation & linking, installation or update, i.e. the program and its data does not change during execution. Dynamic transformations are applied at the same time points as static transformations, however, the program or its data also change during loading or execution. Even though dynamic code transformations are generally considered stronger against MATE attacks than static ones, they require the code pages to be both writable and executable. This opens the door for remote attacks (e.g. code injection attacks [203]), which are more dangerous for end-users than MATE attacks. Moreover, dynamic transformations generally have a higher performance overhead than static transformations, because code has to first be written (generated or modified) and then executed. Therefore, on the one hand, many benign software developers avoid dynamic transformations entirely. On the other hand, dynamic transformations are heavily used by malware developers.

3.2.5. Target of Transformations

The most common dimension for classifying obfuscation transformations is according to the target of transformations. This dimension was first proposed by Collberg et al. [58], who indicated four main categories: layout, data, control and preventive transformations. In a later publication Collberg and Nagra [57] refined these categories into four broad classes: abstraction, data, control and dynamic transformations. Since the last class of Collberg and Nagra [57] (i.e. dynamic transformations), overlaps with the dynamics of transformation dimension, described in Section 3.2.4, we will use a simplification of these two proposals where we remove the dynamic transformations class and merge the abstraction, layout and control classes. Therefore, the remaining transformation targets are:

- *Data transformations*, which change the representation and location of constant values (e.g. numbers, strings, keys, etc.) hard-coded in an application, as well as variable memory values used by the application.
- *Code transformations*, which transform the high-level abstractions (e.g. data structures, variable names, indentation, etc.) as well as the algorithm and control-flow of the

Dimension	Possible values
Abstraction level	Source code
	Intermediate representation
	Binary machine code
Time	Implementation
	Compilation & linking
	Installation
	Loading
	Execution
	Update
Unit	Instruction
	Basic block
	Loop
	Function
	Program
	System
Dynamics	Static
	Dynamic
Target	Data
	Code

Table 3.1.: Classification dimensions for obfuscation transformations.

application.

This dimension is important for practitioners, because it indicates the goal of the defender, i.e. whether the defender wants to protect data or code. Note that obfuscation transformations which target data may also affect the layout of the code and its control-flow, however, their *target* is hiding data, not code. In practice data transformations are often used in combination with code transformations, to improve the potency and resilience of the program against MATE attacks.

3.2.6. Summary of Obfuscation Transformation Classification

Table 3.1 provides a summary of the classification dimensions described above along with the possible discrete values that each dimension can take. In the next section we choose to present a survey of obfuscation transformations classified according to their target of transformation, because it entails a clear partition of transformations.

3.3. Survey of Obfuscation and Diversity Transformations

The following presents a state of the art survey of practical obfuscation transformation techniques, grouped according to their target of transformation, namely data and code.

3.3.1. Data Transformations

Data transformations can be divided into two subcategories, namely *constant data* and *variable data* transformations. In the following we first present an overview of constant data transformations, followed by an overview of variable data transformations.

Constant Data Transformations

Transformations in this category affect static (hard-coded) values. Abstractly, such transformations are encoding functions which take one or more constant data items i (e.g. byte arrays, integer variables, etc.), and convert them into one or more data items $i' = f(i)$. This means that any value assigned to, compared to and based on i is also changed according to the new encoding. There will be a trade-off between resilience and potency on one hand, and cost on the other, because all operations performed on i require computing $f^{-1}(i)$, unless f is homomorphic w.r.t. those operations.

Opaque predicates Collberg et al. [58] introduce the notion of *opaque predicates*. The truth value of these opaque predicates is invariant w.r.t. the value of the variables which comprise it, i.e. opaque predicates have a value which is fixed by the obfuscator e.g. the predicate $x^2 + x \equiv 0 \pmod 2$ is always true. However, this property is hard for the attacker to deduce statically. Collberg et al. [58] also present an application of opaque predicates, which is called *extending loop condition*. This is done by adding an opaque predicate to loop conditions, which does not change the value of the loop condition, but makes it harder for an attacker to understand when the loop terminates.

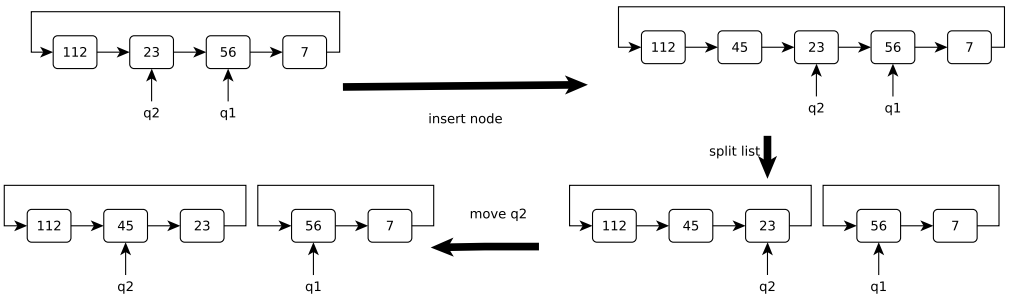


Figure 3.1.: Opaque expressions based on linked lists.

Opaque predicates can be created based on mathematical formulas which are hard to solve statically, but they can also be built using any other problem which is difficult to compute statically, e.g. aliasing. Aliasing is represented by a state of a program where a certain memory location is referenced by multiple symbols (e.g. variables) in the program. Several works in the literature show that pointer alias analysis (i.e. deciding at any given point during execution, which symbols may alias a certain memory location), is undecidable [133, 176, 111]. Therefore, Collberg et al. [58] propose to leverage this undecidability result to build opaque predicates using pointers in linked lists. For instance, consider the linked list illustrated in the top-left part of Figure 3.1. This circular list consists of four elements and it has two pointers (i.e. q_1 and q_2) referencing its elements. After performing three list operations, i.e. inserting another list element, splitting the list in two parts and then moving the pointer q_2 one element forward, the obfuscator knows that the element referenced by q_1 is higher than the element referenced by q_2. However, this relation is hard to determine using static analysis techniques, therefore $q_1 > q_2$ represents an opaque predicate, which is always true. Wang et al. [216] employ such opaque expressions to hide code pointer values, hence, obfuscating control flow via data obfuscation.

One extension of opaque predicates was made by Palsberg et al. [166], who propose *dynamic opaque predicates* which change their truth values between different runs of the program. A further extension appeared in the work of Majumdar and Thomborson [144], who proposed *distributed opaque predicates* which change their truth values during the same execution of a program, depending on the location in code, where they are evaluated.

Convert static data to procedural data (a.k.a. Encode Literals) A simple way of obfuscating a hard-coded constant is to convert it into a function (program) that produces the constant at runtime [58]. This transformation implies choosing an invertible function (program) f, feeding the constant to f as input and storing the output. During runtime the inverse of that function, i.e. f^{-1} is applied to the output of f which was stored somewhere in the program. Obfuscating a hard-coded constant value (e.g. 5), by using simple encoding functions (e.g. $f(i) = a \cdot i + b$), leads to small execution overheads. However, since i is a constant, such functions can also be deobfuscated using compiler optimizations such as constant folding [12]. Therefore, another way of hiding constants is to build expressions dependent on external variables (e.g. user input). For instance, *opaque expressions* – similar to opaque predicates except that their value is non-Boolean – always have a certain fixed value

Listing 3.1: Code before Encode Literals

```
1 int main(int ac, char* av[]) {
2   int a = 1;
3   // do stuff
4   return 0;
5 }
```

Listing 3.2: Code after Encode Literals

```
1 int main(int ac, char* av[]) {
2   double s = sin(atof(av[1]));
3   double c = cos(atof(av[1]));
4   int a = (int) (s * s + c * c);
5   // do stuff
6   return 0;
7 }
```

Listing 3.3: Hiding the value of $k = 0x876554321$ using Mixed Boolean-Arithmetic.

```
int main(int argc, char* argv[]) { // compiled on a 32-bit architecture
 int x = atoi(argv[1]);
 int x1 = atoi(argv[2]);
 int x2 = atoi(argv[3]);

 int a = x*(x1 | 3749240069);
 int b = x*((-2*x1 - 1) | 3203512843);
 int d = ((235810187*x+281909696- x2) ^ (2424056794+x2));
 int e = ((3823346922*x+3731147903+2*x2) | (3741821003 + 4294967294*x2));

 int k = 135832444*d +4159134852*e+272908530*a+409362795*x+136454265*b+2284837645 +
    415760384*a*b+ 2816475136*a*d+1478492160*a*e+3325165568*b*b+2771124224*b*x +
    1247281152*a*x+1408237568*b*d+2886729728*b*e+4156686336*x*x+4224712704*x*d +
    415760384*a*a+70254592*x*e+1428160512*d*d+1438646272*d*e+1428160512*e*e;
 // do stuff
 return 0;
}
```

during program execution, e.g. $cos^2(x)+sin^2(x)$ is always equal to 1, regardless of the value of x. Therefore, the constant value 1 from the C code from Listing 3.1, can be encoded using this opaque expression, which cannot be simplified away by the compiler. The resulting code after this obfuscation is shown in Listing 3.2. This transformation can also be applied to string constants, which can be split into substrings or even single characters, which can be interpreted as integers. At runtime these substrings or characters would be concatenated in the right order to form the original string.

Mixed Boolean-Arithmetic Zhou et al. [232], propose a data encoding technique called Mixed Boolean-Arithmetic (MBA). MBA encodes data using linear identities involving Boolean and arithmetic operations, together with invertible polynomial functions. The resulting encoding is made dependent on external inputs such that it cannot be deobfuscated using compiler optimization techniques. The following example is taken from [232] and it aims to encode an integer value $k = 0x87654321$. The example gives k as an input to the following second degree polynomial with coefficients in $\mathbb{Z}/(2^{32})$:

$$f(x) = 727318528x^2 + 3506639707x + 6132886 \pmod{2^{32}}.$$

The output of computing $f(k)$ is 1704256593. This value can be inverted back to the value of k during runtime by using the following polynomial:

$$f^{-1}(x) = 1428291584x^2 + 1257694419x + 4129091678 \pmod{2^{32}}.$$

Note that Zhou et al. [232] describe how to pick such polynomials and how to compute their inverse. Since the polynomial $f^{-1}(x)$ does not depend on program inputs and the value of $f(k)$ is hard-coded in the program, an attacker can retrieve the value of k by using constant propagation. In order to create a dependency of $f^{-1}(k)$ on program inputs, the

following Boolean-arithmetic identity is used:

$$2y = -2(x \vee (-y - 1)) - ((-2x - 1) \vee (-2y - 1)) - 3.$$

This identity makes the computation of a constant value (i.e. $2y$), dependent on a program input value, i.e. x. Note that this relation can be applied multiple times for different program inputs. The resulting Boolean-arithmetic relation is further obfuscated by applying the following identity:

$$x + y = (x \oplus y) - ((-2x - 1) \vee (-2y - 1)) - 1.$$

Making the computation of $f^{-1}(k)$ dependent on three 32-bit integer input arguments of the program and applying the second Boolean-arithmetic relation multiple times gives the code in Listing 3.3, which dynamically computes the original value of $k = 0x87654321$. Note that in Listing 3.3, variables a, b, d and e are input dependent, common subexpressions of the MBA expression of k.

White-box cryptography This transformation was pioneered by Chow et al. [49, 50], who proposed the first White-Box Data Encryption Standard (WB-DES), respectively White-Box Advanced Encryption Standard (WB-AES) ciphers in 2002. The goal of White-Box Cryptography (WBC) is the secure storage of secret keys (used by cryptographic ciphers), in software, without hardware keys or trusted entities. Instead of storing the secret key of a cryptographic cipher separately from the actual cipher logic, white-box cryptography embeds the key inside the cipher logic. For instance, for Advanced Encryption Standard (AES) ciphers, the key can be embedded by multiplication with the T-boxes of each encryption round [83]. However, simply embedding the key in the T-boxes of AES is prone to key extraction attacks since the specification of AES is publicly known. Therefore, WB-AES implementations use complex techniques to prevent key extraction attacks, e.g., wide linear encodings [226], perturbations to the cipher equations [37] and dual-ciphers [125].

The idea behind the white-box approach in [50] is to encode the internal AES cipher logic (functions) inside Look-up Tables (LUTs). One extreme and impractical instance of this idea is to encode all plaintext-ciphertext pairs corresponding to an AES cipher with a 128-bit key, as a LUT with 2^{128} entries, where each entry consists of 128-bits. Such a LUT would leak no information about the secret-key but exceed the storage capacity of currently available devices. However, this LUT-based approach also works for transforming internal AES functions (e.g. XOR functions, AddRoundKey, SubBytes and MixColumns [83]) to table lookups, which can be divided such that they have a smaller input and output size. Moreover, LUTs can also be used to encode random invertible bijective functions, which are used to further obfuscate the LUTs representing internal AES functions. This leads to an implementation which is much more compact in terms of storage, in the order of a few megabytes.

One-way transformations One-way transformations refer to mapping data values from one domain to another domain (e.g. $f(i) = i'$), without needing to perform the inverse mapping $f^{-1}(i')$ during runtime. This means that f must be homomorphic w.r.t. the operations performed using i. For instance, a cryptographic hash function such a Secure Hash Algorithm (SHA), e.g. SHA-256 (denoted H) may be used as a one-way transformation. H can map a hard-coded string password s to a 256-bit value, i.e. $v = H(s)$. Generally, the only operation performed with a hard-coded password is an equality check with an external user input i. Hence, v does not need to be mapped back to s during runtime. Instead, the program can compute $v' = H(i)$ and verify the equality between the hard-coded value v and the dynamically computed v'. Since the implementation of H for cryptographic hash functions, does not disclose the inverse mapping H^{-1}, the MATE attacker is forced to either guess s, or identify the equality comparison and modify it such that it always indicates equality regardless of i. The latter tampering attack can be hampered if the code of the equality check is highly obfuscated, which can be achieved by applying code obfuscation transformations, as presented in Section 3.3.2.

Variable Data Transformations

The transformations in this category modify the representation or structure of variable memory values. The goal of such transformations is to hamper the development of automated MATE attacks, which can assume that a certain variable memory value will always have a certain representation or a certain structure (e.g. an integer array of 100 contiguous elements). If such assumptions hold then automated attacks are easier to develop, because they mainly rely on pattern matching. Therefore, by employing the transformations presented in this section, one can raise the bar for these kinds of attacks.

Split variables The idea behind this transformation is to substitute one variable by two or more variables [58]. For instance, an integer variable i that only takes values between 0 and 7 can be split into three boolean variables b_1, b_2 and b_3 that represent the integer i in binary format, i.e. $\overline{b_1 b_2 b_3}$. This idea is similar to converting static data to procedural data, except that splitting variables does not apply to constant values, but to any value that a variable may hold at any moment during execution.

Merge variables Two or more variables can be merged into a single variable, if the ranges of the combined variables fit within the precision of the compound variable [58]. For example, up to four 8-bit variables can be packed into a 32-bit variable. Any operations on the individual variables has to be carefully crafted in order not to affect the other variables, which may increase cost. On the other hand, these specially crafted operations stand out to a MATE attacker, who could figure out that multiple variables are stored in a compound variable.

Restructure arrays Similarly to variables, arrays can be split or merged [58]. However, in addition to that, arrays can be folded (increasing the number of dimensions), or flattened (decreasing the number of dimensions). Folding and flattening break code abstractions put in by software developers (e.g. matrices are flattened into arrays), which force reverse engineers to reason about the code in order to recover this useful abstraction.

Reorder variables This transformation targets locations of variables in program memory, by permuting them [53]. It has low cost, but also low potency. The only improvement is in terms of resilience. Pappas et al. [167] apply this transformation at binary level by reassigning register operands at basic block level. They show that this transformation is able to eliminate (on average) over 40% of Return Oriented Programming (ROP) gadgets in different instances of the same program.

Dataflow Flattening Dataflow flattening, proposed by Anckaert et al. [6], is an advanced version of variable reordering proposed by Cohen [53]. It periodically reorders data stored on the heap via a memory management unit, such that the functionality of the program is not altered. In addition to reordering the data on the heap, dataflow flattening also proposes moving all local variables from the stack to the heap and scrambling pointers to hide the relation between the different pointers returned to the program. This transformation has a high potency and resilience, nonetheless, its execution overhead is also high.

Randomized stack frames This transformation assigns each newly allocated stack frame a random position on the stack [86]. Additionally, it pads each stack frame internally by a random amount, such that return address and local variable offsets are located at unpredictable locations. The potency is low, but the resilience is increased.

Data space randomization Bhatkar and Sekar [33] introduce a data transformation technique which they call Data Space Randomization (DSR). The idea of this technique is to XOR data values stored in program memory (e.g. stack and heap) with randomly generated masks. The masks do not need to be fixed, they can be generated dynamically at runtime. The technique is inspired by PointGuard [63], which encrypts code pointers. This technique is reported to introduce an average run-time overhead of 15% and it can protect against buffer- and heap-overflow attacks.

3.3.2. Code Transformations

This section presents transformations, which hide the algorithms (i.e. the logic) inside programs, but also the high-level abstractions added by developers. As opposed to *constant data transformations*, which in some cases perform one-way mappings of data to a completely different domain (e.g. hash functions), code transformations must always perform a

Listing 3.4: Code before Encode Arithmetic

```
1 int main(int ac, char* av[]) {
2   int x = atoi(av[1]);
3   int y = atoi(av[2]);
4   int w = atoi(av[3]);
5   int z = x + y + w;
6   // do stuff
7   return 0;
8 }
```

Listing 3.5: Code after Encode Arithmetic

```
1 int main(int ac, char* av[]) {
2   int x = atoi(av[1]);
3   int y = atoi(av[2]);
4   int w = atoi(av[3]);
5   int z = (((x ^ y) + ((x & y) << 1)) | w) +
6           (((x ^ y) + ((x & y) << 1)) & w);
7   // do stuff
8   return 0;
9 }
```

mapping to the same domain of executable code, because obfuscated code must be able to run on the underlying machine where it is installed.

Instruction substitution This technique (first mentioned in [53]) is based on the fact that in some programming languages as well as in different Instruction Set Architectures (ISAs), there exist several (sequences of) equivalent instructions. This means that substituting an instruction (sequence) with its equivalent will not change the semantic behavior of the program, nevertheless, it will result in a different binary representation. A concrete implementation and evaluation of this technique was first described by Jacob et al. [116] and it is also used in the *Hydan* tool [77]. The transformation has a moderate cost, however, it offers low potency, due to the fact that the number of transformations available is limited. Regarding the resilience of this transformation, Pappas et al. [167] measured the effect of this transformation at binary basic block level, against ROP attacks and discovered that it reduces less than 20% of ROP gadgets. Moreover, the use of uncommon instructions will decrease stealth, i.e. indicate to an attacker where the substitution occurred. In order to improve the stealth of this transform, De Sutter et al. [69] proposed a technique called *instruction set limitation*, which proposes candidates for substitution based on the statistical distribution of instruction types in the program. Mason et al. [146] also proposed a similar technique with the purpose of improving the stealth of shellcode by encoding it as text written in the English language.

Encode Arithmetic This technique is proposed by Collberg [55] and it is a variant of *instruction substitution*, which substitutes boolean or arithmetic expressions by expressions involving both boolean and arithmetic operations, which are harder to understand. One example of such a transformation is illustrated by Listing 3.5, which shows a C code snippet after *encode arithmetic* has been applied to the right-hand side of the assignment to variable `z` from line 5 of Listing 3.4. The drawback of this approach is that there are a limited number of such Boolean-arithmetic identities available in the literature [220]. Eyrolles et al. [79] have proposed writing a reverse transformation for each of them, after identifying MBA expressions via pattern matching.

Garbage insertion This technique implies inserting arbitrary sequences of instructions, that are independent of the data flow of the original program and do not affect its IO behavior (functionality) [53]. The possible sequences that may be inserted are virtually infinite, nonetheless, the performance-cost grows proportionally to the number of inserted instructions. This technique changes the relative offset the original instructions of the program. It also raises the complexity of reverse-engineering by cluttering the original code. However, note that garbage code should be inserted only after performing compiler optimizations, because it can be identified and eliminated via taint analysis. The transformation space for this technique is limited only by physical or practical run-time constraints such as time delays and memory consumption, because as opposed to dead code, garbage code is always executed.

Insert dead code This technique is similar to Cohen's *garbage insertion* technique [53] and Forrest's *adding or deleting nonfunctional code* [86]. Moreover, it also includes the modification of control-flow such that a dead branch is added, i.e. a branch that is never taken during runtime. Adding the dead branch is facilitated by opaque predicates. In order not to disclose the truth value of opaque predicate by leaving the dead branch empty, Collberg suggests inserting dead code (i.e. code that is never executed) on the dead branch. To further confuse the attacker, the dead code can be a buggy version of the other branch, which is always chosen.

Convert a reducible to a non-reducible flow graph This technique is based on the fact that native or machine code can be more expressive than programming languages. This enables *language-breaking transformations* [58], i.e. using instructions from an ISA, which have no direct correspondent construct in the source language (e.g. the goto instruction in Java bytecode has no direct correspondent statement in the Java source language). An automated deobfuscator such as a decompiler will try to find a source language construct that can simulate the lower level instruction, or it will fail. For example, a structured loop is converted into a loop with multiple headers by adding a conditional jump to the middle of the loop, guarded by an opaque predicate will confuse a decompiler.

Program encoding This technique keeps one or more instructions encoded (i.e. encrypted [215, 41] or compressed [164]), while the program is off-line and decodes the sequence(s) when the program is running [53]. The complexity of deobfuscation depends on the algorithm used for encoding, e.g. a compression algorithm can be undone without a secret key, while an encryption requires finding the key. However, the costs may also be relatively high-compared to other techniques, because the code has to be decoded before it can be executed. There is a trade-off between potency, resilience and cost depending on the level of granularity at which this transformation is applied, i.e. if applied at instruction level, the cost as well as potency and resilience are high, while if applied at program level these measures are all low. Additionally, this technique does not protect well against dynamic

analysis attacks, e.g. during execution the code is decoded in memory and it can be read or modified directly in memory by the MATE attacker.

Virtualization obfuscation This technique is related to the *program encoding* technique, because it also implies an encoding of instructions [53]. Additionally, it also requires an interpretation engine (called "simulator" or "emulator"), which is able to decode the instructions and execute them on the underlying platform. Moreover, the simulator may also be running on top of another interpretation engine and so forth, giving an arbitrary nesting level. Normally, this creates complexity for static-analysis attacks, because the attacker has to first understand the custom interpreter logic and then the code running on top of it. The most significant difference of virtualization w.r.t. program encoding is that no code must be written to a memory location during decoding. However, the trade-off between potency, resilience and cost for this transformation are the same as in the case of program encoding.

Self-modifying code This technique has been discussed in several works [53, 141, 123, 147]. It implies adding, modifying and/or removing instructions of a program during its execution. Therefore, it creates a high complexity for static-analysis attacks. Nevertheless, it is not as effective against dynamic-analysis if the executed instructions are exactly the same in different runs of the program with the same inputs. Moreover, if the executed instructions randomly differ from one execution to the other, this technique also offers protection against dynamic-analysis. Similarly to program encoding, this is a dynamic transformation, however, in the case of self-modifying code, there is no dedicated decoder routine. Instead, the existing code is "responsible" for modifying itself and the modifying instructions are often spread throughout the entire program. The trade-off between potency, resilience and cost is similar to that of *program encoding* and *simulation*. Self-modifying code is seldom employed at low units of transformation such as instruction, basic block, loop or function. The most common unit at which it is applied is program level because this leads to a larger set of possible pairs of modifying and modified instructions.

Adding and removing calls This technique first proposed by Cohen [53], can be applied at any unit of transformation. Adding a call to a sub-routine implies: (1) selecting an arbitrary sequence of instructions, (2) creating a sub-routine using that sequence and (3) finally, substituting a call to that sub-routine with the original sequence. Removing a call to a sub-routine implies: (1) substitute the body of a subroutine with all calls to that routine and (2) delete the sub-routine. This causes changes in the structure of a program, which creates more complexity for MATE attacks. The cost of this technique grows or decreases with the number of inserted, respectively removed subroutine calls.

This method has been extended by Banescu et al. [24], such that system calls are added or existing system calls are substituted with equivalent ones. They call this transformation

behavior obfuscation because it hides the system call trace analyzed by behavioral malware analysis engines.

Merging and splitting functions These two techniques are the code correspondents to the data obfuscation transformations of merging and splitting variables [53]. Merging is done by creating larger functions with more inputs and outputs, some of which are independent. Another approach towards merging functions or even basic blocks is proposed by Jacob et al. [117]. They make use of the fact that some ISAs (e.g. Intel x86) have a variable width encoding, which allows the program counter to jump in the middle of instructions and then start interpreting those bytes as the beginning of another instruction. This way two functions or basic blocks can share bytes of code by having their instruction bytes overlapped with each other. The two overlapping functions (basic blocks) have a different starting address and can therefore be executed unambiguously.

Splitting is done by dividing large functions into smaller functions, such that the output of one function are used as inputs to the other. Similarly to the *adding and removing calls* transformation, this technique changes the structure of the program, breaking abstractions added by developers, which makes the code more difficult to understand. Moreover, the cost of this transformation is increased or decreased with the number of split, respectively merged functions.

Loop transformations Several loop transformations have been proposed as compiler optimization passes by Bacon et al. [12]. Collberg et al. [58] argue that these loop transformations also increase software complexity metrics and can therefore be considered obfuscation transformations that increase potency. *Loop tilling* or *blocking* is intended to improve cache locality, by dividing loop iteration lengths into parts that fit in the CPU cache. This increases the nesting level of loops and is therefore more potent. *Loop distribution* or *fission* breaks the independent instructions in a loop body into multiple loops with the same iteration length, which increases the number of loops in the code. *Loop unrolling* replicates the body of the loop a certain number of times and reduces the number of iterations correspondingly, which increases the number of lines of code in the program.

Adding and removing jumps This technique changes the control-flow of the program by adding spurious jumps or removing existing jumps [53]. Adding jumps can be done by substituting an arbitrary sequence of instructions I by: (1) a jump to a random position, (2) followed by I and (3) a jump to the instruction immediately following I in the original version of the program. Removing jump instructions may also be done if it does not alter the original semantics of the program, e.g. unconditional jumps may be removed. However, in practice adding jumps is more frequently employed in order to increase the complexity of the MATE attack. The transformation space of this technique is bounded by the length of the program it is applied to. The cost of this method grows (decreases) with the number of inserted (removed) jump instructions. The potency and resilience of adding jumps can

be increased by further obfuscating the addresses of the jumps using data obfuscation techniques such as *opaque expressions* or *converting static data to procedural data*.

Instruction reordering This technique targets sequences of instructions, which when permuted, do not alter the original program execution [53]. The candidate instruction sequences targeted by this technique are also candidates of parallel processing optimizations, because they can be independently performed by different execution threads, without any danger of race conditions. The reordered sequence of instructions must be equivalent to the original sequence. The cost and potency of this transformation are low. However, Pappas et al. [167] have employed instruction reordering on binary basic block level and have shown that this transformation reduces the number of ROP gadgets by over 30%, hence, increasing the resilience against ROP attacks. Note that this transformation can be also performed at basic block level, however, this would have a lower increase in potency and resilience compared to instruction reordering.

Control flow flattening Wang et al. [216] and Chow et al. [51] proposed Control Flow Flattening (CFF), which collapses all the basic blocks of a function into a flat (3-level) Control Flow Graph (CFG), which hides the original control flow of the program. The first level of the CFG is similar to an interpreter dispatcher, which chooses the right basic block where to go on the second CFG level. The third level of the CFG is a common exit point for all basic blocks on the second levels, which loops back to the basic block on the first CFG level. Such a 3-level CFG may be implemented using a *switch* statement embedded inside an infinite loop. The order in which the cases of the switch statement must be executed is indicated by an integer (control) variable, which is updated by every case of the switch statement accordingly, before, during or after part of the logic of the original program is executed. The infinite loop is exited when a case contains a return or break statement.

Branch functions Linn and Debray [139] propose hiding the control flow of calls, conditional and unconditional jumps, from static disassembly algorithms, by replacing them with calls to a so called *branch function*. A branch function computes the actual target of the jump dynamically using a parameter passed by the callee. Instead of returning to the instruction immediately following the call instruction, the branch function either jumps to the address of the original jump instruction which it replaced, or to several "junk" bytes after the call instruction that it replaced. The structure of the control flow graph is also flat similar to CFF, however, the *switch*-statement is replaced by the branch function.

Schrittwieser and Katzenbeisser [183] present an extension of branching functions, which is explicitly aimed at defending against both static- and dynamic-analysis techniques. The targets of the branch functions are ROP gadgets (i.e. short instruction sequences ending in a return instruction) [188]. Additionally, they generate gadget graphs which add redundancy such that the one path in the original code can have multiple paths in the obfuscated code. This is meant to hamper dynamic analysis attacks by generating different traces for the

same inputs. The disadvantage of this method is that its potency and resilience increases inversely proportional to the size of the gadgets, while its cost decreases exponentially with this size.

Remove comments and change formatting This transformation is only applicable to programs which are delivered as source code (e.g. JavaScript). Comments are removed if they exist and all space, tab and newline characters are also removed, which results in a continuous string of code which is more potent against human attackers, than the original code. The original formatting cannot be recovered [58]. The cost of this transformation is also low and in many cases it even improves memory costs and execution speed. However, a similar alignment can be automatically generated via static analysis. Therefore, the resilience of this transformation is very low. However, they have found their way into commercial products, such as: Stunnix [198], DashO [172], Dotfuscator [173], Thicket [186], ProGuard [99] and yGuard [230].

Scrambling identifier names This transformation implies changing all symbol names (e.g. variables, constants, functions, classes, etc.) into random strings [58]. This is a one-way transformation, because the names of the symbols cannot be automatically recovered by a deobfuscator. Therefore, the MATE is forced to understand what a symbol is from a semantics point of view. It has a much higher potency and resilience than formatting removal since identifiers contain useful abstractions added by software developers. Similarly to removing comments and changing formatting, this transformation has a very low cost and it is also used as an optimization that reduces code size, because long symbol names can be replaced by shorter ones.

Removing library calls and programming idioms Most programs perform calls to external libraries providing useful data structures (e.g. lists, maps, etc.) and algorithms (e.g. sorting, searching, etc.). MATE attackers often start by inspecting calls made to external libraries to give a high-level indication of what the program is doing. This transformation implies replacing such dependencies on external libraries with own implementations where possible [58]. Note that such a transformation is stronger than static linking, which only copies the code of the library routines in the executable. Static linking can be easily reverse engineered by pattern matching attacks [195]. Techniques from the field automatic program recognition [222] can be used to identify common programming patterns and replace them with less obvious ones. For example, consider iterating over a linked list; the standard list data structure can be replaced with a less common one, such as cursors into an array of elements.

Modify inheritance relations Programs written in some object-oriented programming languages are distributed in some intermediate format to end-users (e.g. C#, Java, etc.). These intermediate formats are only compiled to native code on the client's machine

and contain useful object-oriented programming abstractions. In such programs it is important to break the useful abstractions offered by classes, their structure and their relations (e.g. aggregation, inheritance, etc.). According to Collberg et al. [58], the complexity of a class grows with its depth in the inheritance hierarchy and the number of its direct descendants. This can be done by splitting classes and inserting dummy classes. One variant of class insertion is called *false refactoring* [58]. False refactoring is performed on two or more classes that have no common behavior. All instance variables of these classes having the same type are moved into the new parent class. Methods of the parent class can be buggy versions of methods from its child classes. This approach has been further extended by Foket et al. [85], who propose a technique called *class hierarchy flattening*. In this approach a common interface that contains all methods of all classes is created. All classes implement this common interface and they have no other relationship between each other. This effectively destroys class hierarchies and forces the attacker to analyze the code.

Function argument randomization Randomizing the order of formal parameters of methods and inserting bogus arguments is a technique implemented by tools such as Tigress [56]. The purpose of this transformation is to hide common function signatures across a large diverse set of instances. This transformation is straightforward to perform for programs which do not offer an external interface (e.g. libraries). However, if this obfuscation is applied to a library, then it changes the interface (of that library), and all the corresponding programs using that library will have to be updated as well. The potency, resilience and cost of this transformation are low.

3.3.3. Summary of Survey

In the survey presented above, we have enumerated several practical data and code obfuscation transformations. On the one hand, these transformations can be applied to real-world software applications, as opposed to cryptographic obfuscation, presented in Chapter 2. On the other hand, practical obfuscation does not offer provable security guarantees like cryptographic obfuscation does. Nevertheless, many contexts mandate the use of practical obfuscation transformations to protect digital software assets, e.g. secret keys, premium content, intellectual property of code, etc. In such contexts, the goal is to raise the bar against the majority of MATE attackers, not all possible attackers, e.g. developers are concerned about malicious end-users, not governmental organizations, which are highly funded. Therefore, it is crucial for software developers to determine which software protection techniques to use in order to achieve this goal. The following chapters of this thesis describe a framework which enables a characterization of software protection strength, by means of computing the effort against automated MATE attacks. This framework aids software developers in choosing the appropriate obfuscation transformations for their applications.

Part II.

The Core

4. Automated MATE Attacks

This chapter presents the main contribution of this thesis: a model for reasoning about obfuscation strength by representing different steps of all automated MATE attacks as search problems. Parts of this chapter have previously appeared in a peer-reviewed publication [21], co-authored by the author of this thesis.

Man-At-The-End (MATE) attacks can be of two types: human-assisted or automated. *Human-assisted attacks* require non-trivial guidance by a human agent, who has experience and intuition regarding the attack. Human-assisted attacks involve a trial-and-error approach to problem solving and often require creativity for reaching the goal. On the other hand, *automated attacks* involve a programmatic approach for reaching the goal, i.e. all of the information and steps required by the attack are known and well defined such that they are or can be implemented using a combination of software and hardware.

Human-assisted attacks

Automated attacks

As discussed in Chapter 1, this thesis focuses on automated attacks, because human-assisted attacks, do not scale when a large number of diverse software instances are targeted by a MATE attacker. Moreover, the effectiveness of obfuscation against human-assisted attacks is difficult to quantify, since it not only depends on the features of the tools used by the MATE attacker, but also on the knowledge and skills of that attacker. For instance, given the same deterministic obfuscated program and the same analysis tools (e.g. IDA Pro [73]), a professional malware analyst is highly likely to complete the task of analyzing the program, significantly faster than a computer science student who is familiar with the analysis tools, but has little experience with obfuscated programs. On the other hand, executing an automated attack multiple times – with the same seed, if the attack is randomized – on the same obfuscated program, results in the same analysis time – minus a negligible delta due to other background processes and the OS scheduler – regardless of the skills of the human executing the attack.

4.1. Classification of Automated MATE Attacks

In contrast to the classification of software protection techniques (see Chapter 3), the classification of MATE attacks has been the topic of relatively few publications [54, 1, 184]. This section presents the most relevant classification dimensions w.r.t. the contents of this thesis.

4.1.1. Attack Type Dimension

Basile et al. [30] argue that it is not feasible to consider every possible attacker goal since it represents the desired end-result for the attacker, e.g. see the position of other players in computer games, or play premium content without paying, etc. Therefore, in this thesis we classify attacks according to their type, i.e. the means through which an attacker goal can be achieved. According to Collberg et al. [55, 54], there are four types of information a MATE attacker may be interested in recovering from an obfuscated program:

- The original or a simplified version of the *source code*. This is always the case for MATE attackers who are interested in intellectual property theft, i.e. stealing a competitor's algorithm.
- A statically embedded or dynamically generated *data item*. Common examples of such data items are decryption keys used by DRM technologies to play premium content only on authorized devices, for authorized users. However, data items may also include hard-coded passwords, IP addresses, etc.
- The sequence of obfuscation transformations and/or tools used to obfuscate (protect) the program, also called *metadata*. This kind of information is often used by antivirus engines to detect suspicious binaries, based on the fact that several previously seen malware have used the same obfuscation transformations and/or tools.
- The *location*, (i.e. lines of code or bytes) of a particular function of the code. For instance, the attacker may be interested in the module which performs premium content decryption in order to copy it and reuse it in another program, without necessarily understanding how it works.

We compare these four information types proposed by Collberg et al. [55, 54], with the *analyst's aims* proposed by Schrittwieser et al. [184]:

- *Code understanding*, which according to its description in the paper maps to the *source code* information type described above. However, the name of this category suggests a more general attack type than a full recovery of the entire *source code*, because it could be sufficient to have a partial code understanding. For example, a malware analysis engine can decide that a software is malicious using its observed behavior (e.g. unsolicited calls to premium telephone numbers), which does not require full understanding of the source code. Moreover, *metadata* recovery also falls inside of this category of *code understanding*. Therefore, in this thesis we will use this more general category, i.e. *code understanding*.
- *Finding the location of data*, which maps perfectly onto the *data item* information type described above. However, the phrase *location of data* may be mistaken for the *location* information type. Therefore, this thesis will simply use *data item recovery*.

- *Finding the location of program functionality*, which maps onto the *location* information type described above. However, Schrittwieser et al. [184] also add that this type of information may be used to answer questions regarding if the program is malicious or not. In this thesis we move such questions to the *code understanding* category, because we believe an answer to such a question, requires some level of code understanding, but not necessarily recovering the entire *source code*.

- *Extraction of code fragments*, which does not directly map onto any of the information items described above. However, we believe that *finding the location of program functionality* is a prerequisite to this aim of the analyst, because extraction can only be done after the location of the code fragment has been recovered. Therefore, in this thesis we associate this aim of the analyst with the *location* information type.

In sum, we use the following three categories of attack types with the meanings discussed above: (1) *code understanding*, (2) *data item recovery* and (3) *location recovery*.

4.1.2. Dynamics Dimension

Dynamics is one of the most commonly used classification criteria for automated MATE attacks and it refers to whether the attacked program is executed on a machine or not, i.e.:

- *Static analysis* attacks do not execute the program on the underlying (physical or virtual) machine. The subject of the analysis is the static code of the program.

- *Dynamic analysis* attacks run the program and record executed instructions, function calls and/or memory states during execution, which are the subject of analysis.

Static attacks are commonly faster than dynamic attacks. However, static analysis attacks do not handle many code obfuscation transformations as well as dynamic analysis attacks. This does not mean that dynamic analysis attacks can handle any kind of obfuscation easily. For instance, it is challenging to dynamically analyze programs employing code transformation techniques that achieve *temporal diversity*, i.e. the program has significantly different execution traces and/or memory states on every execution. Moreover, dynamic analysis is in general incomplete, i.e. it cannot explore or reason about all possible executions of a program, as opposed to static analysis.

4.1.3. Interpretation Dimension

Code interpretation refers to whether the program's code or the artifacts generated using it (e.g. static disassembly, dynamic traces), are treated as text or are interpreted according to a semantic meaning (e.g. operational semantics). Therefore the two types of code interpretation considered in this thesis are:

- *Syntactic attacks* which treat the program's code or any other artifacts generated by executing or processing it, as a string of bytes (e.g. characters). For example, pattern matching on static code [195] and pattern recognition via machine learning traces of instructions [24], treat the code as a sequence of bytes.

- *Semantic attacks* which interpret the code according to some semantics, e.g. denotational semantics, operational semantics, axiomatic semantics and variations thereof [84]. For example, abstract interpretation [62] uses denotational semantics, while fuzzing [200] uses operational semantics.

Syntactic attacks are generally faster than semantic attacks due to the missing layer of abstraction that interprets the code. Coincidentally, most syntactic attacks are performed via static analysis and most semantic attacks are performed via dynamic analysis. However, there are exceptions e.g. the syntactic analysis of dynamically generated execution traces and semantic static analysis via abstract interpretation, which will be presented in more detail in Section 4.3.

4.1.4. Alteration Dimension

Alteration refers to whether the automated MATE attack changes (alters) the code or not. This type of classification is analogous to the *message alteration* classification of MITM attacks on communication channels. Hence there are two types of code alteration:

- *Passive attacks* do not make any changes to the code or data of the program. For instance, extracting a secret key or password from a program does not require any code alterations.

- *Active attacks* make changes to the code or data of a program. For example, removing data or code integrity checks (e.g. password checks), requires modifying the code of the program. Also "disarming" malware may also involve tampering with its code.

If an attacker's goal can be achieved via either passive or active attacks, then the type of attack used depends on the complexity of the program under attack and the types of protection the program has in place. For instance, if a program is protected via dynamically verified checksums of the program input, then active attacks require finding and disabling these checksumming instructions, which could be more costly than a passive attack.

4.1.5. Summary of MATE Attack Classification

Table 4.1 provides a summary of the classification dimensions described above along with the possible discrete values that each dimension can take. In the remainder of this thesis we will refer to these dimensions when describing attack implementations.

Dimension	Possible values
Attack type	Code understanding
	Data recovery
	Location recovery
Dynamics	Static
	Dynamic
Code interpretation	Syntactic
	Semantic
Alteration	Passive
	Active

Table 4.1.: Classification dimensions for automated MATE attacks.

4.2. Definition of Automated MATE Attacks

Intuitively, an automated MATE attack as a collection of software applications, which are linked together through inputs and outputs and which are able to satisfy the goal of a MATE attacker. A MATE attacker aims to compromise the confidentiality, integrity or availability of data or code in a software application, e.g. recovering a secret key from one or more applications, modifying the behavior of one or more applications, etc. Therefore, we model an automated MATE attack using a path in an *attack net* [150]. An *attack net* is a *petri net* [158], where *places* represent digital items (i.e. data or code) and *transitions* represent execution steps of an automated MATE attack. Places are connected to transitions via directed arcs. The attack net has one or more starting places (sources), i.e. places which have no incoming arcs, and one or more ending places (sinks), i.e. places which have no outgoing arcs. An automated MATE attack is started by assigning a set of tokens to a subset of sources, such that one or more transitions can fire. Tokens are consumed by transitions and each transition execution produces a token. The attack ends when one of the sinks contains a token. If we run all paths in parallel, the best (e.g. fastest) attack will generate the first token in the sink. In practice, a MATE attacker would prefer to pick the best attack path, instead of running all attacks in parallel. However, the attacker cannot determine the best attack path without, at least, estimating the effort of each transition along each path.

Attack net

For instance, consider the attack of bypassing a license check in a software application such as a computer game. An example of the *attack net* corresponding to this attack is shown in Figure 4.1, where places are represented by ovals and transitions are represented by rectangles. Each step (i.e. transition) in this attack net is an algorithm with inputs and outputs. Each of the four possible first steps of this attack net take the binary executable code as input and each of the last steps outputs the digital item necessary to achieve the attacker's goal. In Figure 4.1 there are two possible sinks, i.e. (1) the attacker obtains the license key without paying for it, or (2) the binary code is patched such that there is no need for a license key anymore. There are five different possible paths that can be taken by the MATE attacker from the source to one of the sinks:

1. The path at the top of Figure 4.1 shows an automated MATE attack where the license

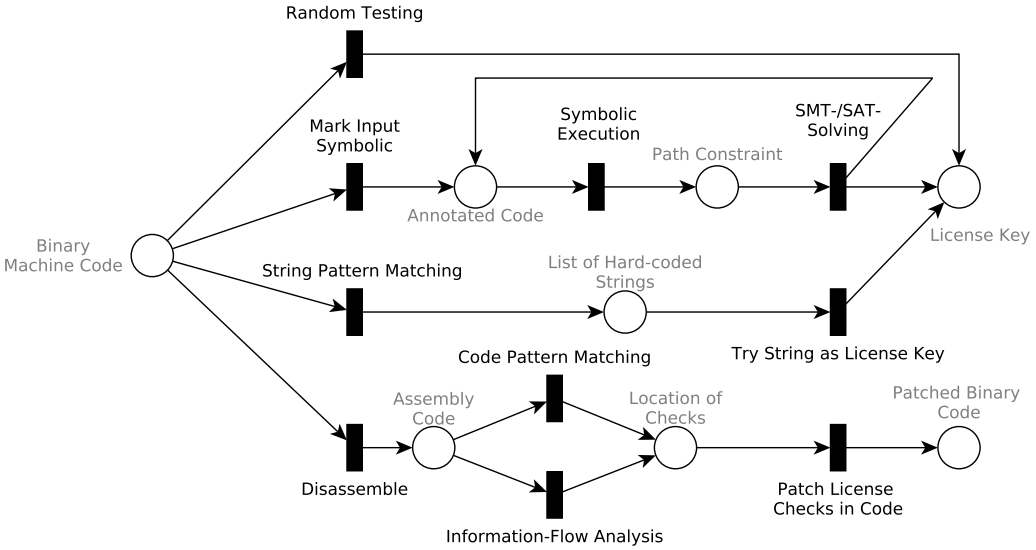


Figure 4.1.: Attack-net representing automated MATE attack for bypassing license check.

key is found via random testing the binary code, i.e. enumerating different program inputs. According to our taxonomy from Section 4.1, this attack can be classified as: *type* = data item recovery, *dynamics* = dynamic, *interpretation* = semantic and *alteration* = passive. Further details about this attack are given in Section 4.3.2.

2. The second path in Figure 4.1 shows an automated attack where the license key is extracted by first marking the input corresponding to the license key as symbolic and then symbolically executing the code to gather path constraints (see Chapter 5). Each path constraint is sent to a SAT-/SMT-solver, which either returns concrete values to satisfy the constraints, or it returns that no such values were found. If sufficient concrete values are found, symbolic execution continues until it finds a path constraint equivalent to the license check. If this path constraint can be solved, by the SAT-/SMT-solver, then the attacker obtains a correct license key. Note that this attack has multiple steps (i.e. transitions) and different steps may be classified in different categories of our taxonomy from Section 4.1. For instance, the first and third steps on this path are static, while the second step is dynamic. If this is the case, then we will classify the entire path as dynamic. All the other classification criteria are the same as for the random testing attack path.

3. In the third attack path from Figure 4.1 all the hard-coded strings are extracted from the binary in the first step. In the second step each of these strings is tried as an input to the program, until all the strings are exhausted or the license key is found. The classification of this attack is slightly different from the previous ones, i.e. *type* = data item recovery, *dynamics* = static, *interpretation* = syntactic and *alteration* = passive.

4. In the fourth path from Figure 4.1 the binary code is first disassembled and then a pattern matching is applied to find the location of the license check(s). Finally, the check is patched such that the binary executable no longer performs any license checks. Note that the first two steps of this attack are passive and the last step is active. Also the first step is semantic, while the second and third steps are syntactic. Therefore, we classify the entire attack as: *type* = location recovery, *dynamics* = static, *interpretation* = semantic and *alteration* = active.

5. The last path from Figure 4.1 shows an automated MATE attack similar to the previous path, where the code is first disassembled. Afterwards the location of the license checks is determined via information-flow analysis. Finally, the checks are patched similarly to the previous attack path. This attack path is classified in the same way as the previous attack. Note that in this case the second step of the attack is also semantic. Further details about this attack are given in Section 4.3.2.

The overarching goal of this thesis is to characterize the strength of obfuscation. This is achieved by characterizing the effort needed by different automated MATE attacks (as the ones enumerated above), on obfuscated software. Therefore, we must be able to characterize each of the steps (i.e. transitions) on a path from a source to a sink. One the one hand, some of the transitions (e.g. *Mark Input Symbolic* or *Patch License Checks in Code*), may be trivial to characterize, because they are deterministic, i.e. the output of the transition is the same across multiple executions of that transition with the same input value. On the other hand, other transitions are not as trivial to characterize (e.g. *Symbolic Execution*, *SAT-/SMT-Solving*), because they involve non-deterministic search. This is the key observation of this thesis, namely that *all automated MATE attacks involve one or more search problems*, which are not trivial to characterize. However, not all problems are search problems. In the example form Figure 4.1 all paths from source to sink imply one or more *transitions which are solved by searching*, e.g. a search for the actual value of the license key or a search for the code implementing the license check. In Section 4.2.2 we propose a search model, which can be used to characterize the cost of each transition that implies search.

4.2.1. Formalization of Automated MATE Attacks

Code is represented by a sequence of bytes at syntactic level, however, at a semantic level it consists of three types of information, which are valuable for a MATE attacker. The three different information types stem from the *attack type* dimension presented in Section 4.1.1, and they are denoted by:

- τ, which represents the logic/algorithm implemented by (a part of) the code. This logic/algorithm can be represented using any level of abstraction (e.g. pseudo-code, first order logic, etc.). This information type maps to *code understanding*.
- δ, which represents non-executable data hidden inside the code. This information type maps to *data recovery*.

- λ, which represents a location of one or more lines of code, which together perform a certain function (e.g. a license check, a code integrity check, etc.). This information type maps to *location recovery*.

Therefore we define code as the Cartesian product of these three information types, i.e.:

$$Code \subseteq Seq(\tau) \times Seq(\delta) \times Seq(\lambda),$$

where $Seq(x)$ indicates a sequence of information items having type x. Given this definition of code, an obfuscator that applies one or more transformations presented in Chapter 3, is a function: $obf_x : Code \rightarrow Code$, where $x \in 2^{\{\tau,\delta,\lambda\}}$, i.e. an obfuscator aims to hide any combination of: logic, data and locations. Not that if $x = \emptyset$, then $obf_\emptyset$ represents the identity transformation. The previous definitions allow us to formalize an automated MATE attack as finding an "inverse" function obf_y^{-1} such that:

$$x \in 2^{\{\tau,\delta,\lambda\}}, y \in \{\tau,\delta,\lambda\}, c \in Code : obf_y^{-1}(obf_x(c))|_y = c|_y,$$

where $|_y$ represents the projection of $c \in Code$ to $y \in \{\tau,\delta,\lambda\}$. This means that even if the obfuscator aims to hide the set of information types x (e.g. logic and location by applying virtualization obfuscation), the MATE attacker is interested in recovering one information type y (e.g. location, because s/he wants to disable license checks in the code).

The challenge for the MATE attacker is that s/he only has a *specification* for obf_y^{-1}, namely $obf_y^{-1}(obf_x(c))|_y = c|_y$, but no *implementation*. Hence, the MATE attacker needs to approximate the implementation by search. Note that the MATE attacker does not know the exact value of $c|_y$ before the attack, but all MATE attacks are associated with a goal function $g : \{Seq(\tau), Seq(\delta), Seq(\lambda)\} \rightarrow \mathbb{R}^+$, such that:

$$g(c|_y) = 0 \wedge \forall c' \in Code, c'|_y \neq c|_y : g(c'|_y) > g(c|_y).$$

Function g is the function that indicates that the automated MATE attack has reached its goal and the search can stop. The output of g may indicate an ordering relation between different $c', c'' \in Code$, e.g. when removing sequences of dead code from code, g may indicate the number of dead instructions still present in the code. However, g does not necessarily indicate an ordering relation between $c', c'' \in Code$, i.e. $g(c'|_y) < g(c''|_y)$ does not necessarily mean that $c|_y$ is somehow more similar to $c'|_y$ than it is to $c''|_y$. For instance, when searching for a hard-coded symmetric key k inside $obf_\delta(c)$, in order to decrypt a media stream m encrypted with k, denoted $e = enc_k(m)$. The MATE attacker does not know m, however, s/he knows that the correct key k will result in a decryption of e – denoted $dec_k(e)$ – which has H.264 encoding and multiple frames have a high amount of green in the bottom half, due to the color of the grass on the pitch. Therefore, g is defined as the number of non-green pixels – in the bottom half of the frame – dived by the number of (all shades of) green pixels, truncated to an integer. This definition of g indicates that if more than half of the pixels in the bottom half of the frame are green, then the correct decryption key has been found. Using the wrong decryption keys (e.g. k' and k'') leads to garbage content

being decrypted, which highly likely results in non-zero values when applying g (e.g. 3 and 5, respectively). One cannot say that one of these keys is closer to the correct key k, because its decrypted frame contains more green pixels.

If the MATE attacker had the implementation of obf_y^{-1}, then s/he could easily retrieve the hidden information $c|_y$. However, explicitly computing obf_y^{-1} is in general impossible, because even if the MATE attacker knows obf_y, s/he cannot compute its inverse. For instance, when renaming variables, applying one-way transformations or even asymmetric cryptography (when the private key is not known), it is not feasible to compute their inverse transformations. Therefore, the only viable alternative for the MATE attacker is to approximate obf_y^{-1} by cleverly enumerating (i.e. searching for) $c'|_y$ such that:

$$g(c'|_y) = 0$$

This search can be performed statically on the obfuscated program $obf_x(c)$, e.g. when searching for data in the code that corresponds to the license key. The search can also be performed dynamically, e.g. when searching for an input i of $obf_x(c)$, which leads the program to execute a certain instruction.

From a pragmatic perspective, it does not make a difference if the attack finds secret data by enumerating (i.e. searching for) inputs to the code or by enumerating code, because the MATE attacker will always pick the fastest way. However, for the defending party, who wants to apply software obfuscation transformations to protect code and/or data, it is important to be able to estimate the complexity of the attack and identify code characteristics, which transformed by obfuscation, increase the time of an automated MATE attack. In Section 4.2.2 we propose a search model to aid the defending party in this way. By being able to approximate the complexity of an automated MATE attack, the defender knows if additional defenses are required or not. Using our framework this can be done without spending the resources to actually run the automated MATE attack, which could cost days or months of computation time.

4.2.2. Search Model

Before presenting the search model we define the terms needed to understand this model. The *goal* of automated MATE attackers is a *digital item* (i.e. data or executable code) having a specified property, which must be recovered or obtained from another digital item. This property of the goal is given by the definition of the goal function mentioned in Section 4.2.1. Examples of goals – inspired from Figure 4.1 – include: (1) *an input value for the license key input of software X, which enables feature Y*, or (2) *a patch of one or more locations in software X, which enables feature Y*, or even (3) *assembly code of software X*. Moreover, each place (i.e. input and output of a transition), in the attack-net from Figure 4.1 contains instances of digital items. Digital items in automated MATE attacks can be categorized as: (i) parts of the execution environment where the program is running (e.g. file-system, process-memory), (ii) the binary or packaged code of the the target program, (iii) the program's input space and (iv) any digital items generated from or by the program (e.g. CFG, instruction traces).

Goal

#	Data struct.	Digital item	Initial state	Action(s)	Goal state
1	String	Program input	Empty string	Add or remove characters to string	Input string causing segmentation fault
2	Array	Binary or packaged code	No strings marked in code	Move index back and forth, mark as string	Set of hard-coded strings marked in code
3	Set	Set of hard-coded strings	Random element from set	Remove element from set	Hard-coded password
4	List	Disassembled or decompiled code	No instructions marked	Process or mark instruction	All integrity checks marked
5	Graph	Control-flow graph	Entry point node	Add or remove nodes (basic-blocks)	Shortest path from entry point to block that outputs error

Table 4.2.: Search problem specification examples

Problem

A *problem* is defined by a given input digital item and a desired output digital item, which is the *goal of the problem*. A problem can be illustrated using the input and output places of a transition in an attack net. The goal of a problem is not necessarily the goal of the MATE attacker, because the goal of one problem, is used as the input of the subsequent problem in the path from source to sink in the attack net. For a given attack-net the digital items in the sink represents the goal of the MATE attacker. To reach their goal, MATE attackers must solve one or more problems. Solving problems is done by executing an *algorithm*, which is a computational process involving a set of instructions that must be followed in a specified order. A *solution* to a problem is a trace of each execution step of an algorithm, starting from the input data and leading to the goal of that problem. In addition to finding a solution, an algorithm also applies the solution to the input digital item of the problem and gives the output digital item, instead of the solution itself. Therefore, an algorithm can be illustrated using a transition in an attack net. Note that, at an abstract level any digital item is a *data structure*. Some examples of digital items and their corresponding data structures are shown in the first two columns of Table 4.2. Treating inputs of algorithms as data structures, has the advantage that we can analyze the effort of such algorithms – given a certain input – as a function of the characteristics of these data structures. For some problems, it is trivial to quantify the effort needed to find their solutions (e.g. the *Make Input Symbolic* transition in Figure 4.1), because this is done by a deterministic algorithm. For other problems, the process of finding a solution involves a non-deterministic search algorithm, which is less trivial to quantify. Therefore, most of the effort of automated MATE attacks is spent solving non-deterministic search problems. A *search problem* is a type of problem that can be solved by a computational process called a *search algorithm*. Before solving a search problem using a search algorithm, one must choose a certain level of abstraction to define the following necessary elements, which represent building blocks for defining the search problem and algorithm:

Algorithm

Solution

Data structure

Search problem

Search algorithm

State

- A *state* is a digital item (having a data structure $d \in D$), which may be annotated by marking certain elements of the data structure. Markings are denoted as $m \in M$. For clarity, we will use a *running example*, where the digital item representing a program

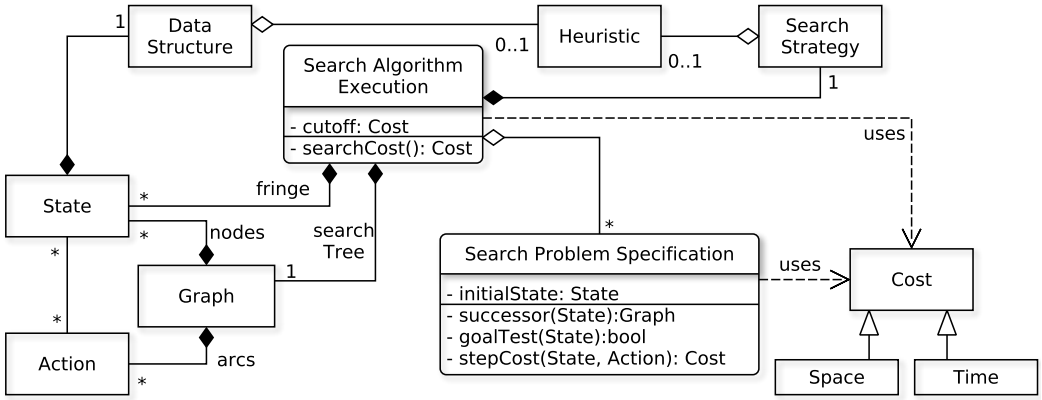


Figure 4.2.: Abstract UML model of search.

(i.e. a list of instructions), is annotated with the current instruction pointer. A state may also be associated with one or more auxiliary digital items. In our running example the auxiliary digital item associated to a list of instructions, is an array of bytes representing the memory of the program. All states that are associated with the same problem belong to a set of states $S \subseteq D \times Seq(M)$.

- An *action* is a transition function between a pair of states, i.e. if A is the set of all actions associated to the current search problem, then $\forall a \in A, a : S \to S$. Each state in S has a known set of actions, each leading to another state. Actions also have a semantic meaning, as shown in the fourth column of Table 4.2. In our running example each state has the possible action of executing the instruction immediately following it, or in case of a jump instruction, the instruction where the control-flow jumps to. *Action*

According to the hypothesis of this thesis, characterizing the strength of different obfuscation transformations against an automated attack is equivalent to characterizing the effort needed by a search algorithm to solve a search problem. To facilitate the characterization of the effort needed by a search algorithm to solve a search problem, we propose a model (illustrated in Figure 4.2[1]), involving a static (fixed) part, called the *search problem specification* and a dynamic part, called the *search algorithm execution*, which will be defined next. The *search problem specification* consists of the following elements: *Search problem specification*

1. An *initial state* ($s_0 \in S$), which is the first state of the search algorithm execution (see examples in column three of Table 4.2). In our running example the initial state is given by annotating the program's entry point as the current instruction and initializing the digital item representing the memory of the program with all zero values. *Initial state*

2. A *successor* function ($suc : S \to 2^{A \times S}$), which returns the set of all actions that can be performed in a given state and indicates the states where these actions lead to. In our *Successor*

[1] We have used terms similar to those in Chapter 3 of a textbook on Artificial Intelligence [178]

running example we obtain the successor function by applying an action to a state, e.g. executing the instruction indicated by the current state will lead to a state where the instruction pointer is moved to the subsequent instruction and the memory is updated according to the operational semantics of the executed instruction. Therefore, one can derive the successor function using the definition of all possible actions.

Goal test

3. A *goal test* function ($g : S \to \mathbb{R}^+$), which indicates the distance of the state given as input to the goal, i.e. if $s \in S$ and $g(s) = 0$ then s is a goal state. The distance from the goal has no standard measurement unit, i.e. it depends on the search problem. For some problems, $g(s)$ only has two possible outputs (i.e. a Boolean output), namely 0 if s is a goal state and 1 if s is not a goal state. Examples of states where a goal was reached are given in column five of Table 4.2. In our running example, we choose the goal state as the state where the current instruction pointer is set to a target instruction that prints a distinctive message (e.g. "You win"), on standard output. Therefore, the goal test is computed by counting the number of instructions between the current instruction pointer and the target instruction.

Cost

4. A set of *cost* metric units (C), which quantify the effort of all actions. Cost metric units include various aspects of *time* (e.g. CPU hours) and *space* (e.g. bytes). In our running example, a cost metric could be the number of CPU hours.

Step cost

5. A *step cost* function ($c : S \times A \to C$), which quantifies the cost of taking a certain action while in a certain state (see column six of Table 4.2). In our running example, the step cost is the the number of CPU cycles needed to execute an instruction, move the instruction pointer and update the memory.

Search algorithm execution

The *search algorithm execution* requires a search problem specification. Additionally, it also consists of the following elements:

Search tree
Expanded

1. A *search tree* $T = (S_s, A_s)$, where S_s and A_s are multi-sets containing elements from S and A, respectively. The search tree is rooted in the initial state and it is *expanded* by applying the successor function to this state, i.e. $suc(s_0)$. This expansion consists of adding the arcs (i.e. actions) and nodes (i.e. states having a unique identifier) – returned by the successor function – to the search tree. Note that each node in the search tree has a unique identifier, which allows multiple nodes to contain the same state in the search tree. *It is important to note that the successor function does not impose any order on its output set of (action, state) pairs*. Therefore, a search algorithm cannot pick the "left-most" or "right-most" node from the set of successors. Instead, any node – from the set of successor nodes – is selected nondeterministically by a search algorithm. Expansion of the search tree continues by applying the successor function to other non-expanded nodes. There are no cycles in a search tree, which means that there can be multiple nodes in the search tree representing the same state. Therefore, the search tree holds the *history* of the search algorithm execution.

2. A *fringe* ($F \subseteq S_s$) is the set of nodes in the search tree which have not been expanded yet, i.e. they are leaves of the tree. Intuitively, the fringe is an annotation on the search tree which indicates the leaves of the tree. In each step of the search algorithm execution a node on the fringe is expanded. *Fringe*

3. Zero, one or two *heuristic* functions ($h \in H$). Heuristic functions may be associated with: (1) the *search strategy* (see next bullet point for definition) or (2) the *data structure* representing a state of the problem. If associated with the search strategy, heuristics are defined as $h : S \to C$, and they prioritize the states on the fringe by estimating their utility. If associated with the data structure, heuristics are defined as $h : D \to D$, and they transform the data structure and hence the state space of the problem from S to S' such that the size (breadth and/or depth) of the search tree is smaller and/or the cost of computing the g function is smaller. Such heuristics are generally applied once, before starting the search algorithm execution (see example in Section 4.3.1). Heuristic functions are derived from information about the search problem ($i \in I$), e.g. side channels. Heuristic function derivation is generally a human-assisted process, mathematically represented by the function $drv : I \to 2^H$. Note that a search execution may use: (1) no heuristic, (2) one heuristic associated with the search strategy, (3) one heuristic associated with the data structure or (4) two heuristics, where one is associated with the search strategy and the other with the data structure. *Heuristic*

4. A *search strategy* function ($stg : 2^S \times H \to S$, where $stg \in A$), which selects a state from the fringe of the search tree, optionally using a heuristic function. A common strategy is to select the best state from the fringe, i.e. the greedy strategy. However, often selecting the best state is not possible, because after applying the heuristic to all states on the fringe, there are multiple states which have the same (highest) utility value. In such cases – where there are multiple "best" states on the fringe – the strategy is used to pick one state. Examples of strategies which do not use heuristic functions include: *breadth-first search*, *depth-first search* and *iterative deepening depth-first search*. *Search strategy*

5. A *search cost* function ($t : 2^{S \times A} \to C$), which quantifies the time and space of the search algorithm execution. In order to do this the search cost function sums up: (1) the step costs of all the (state, action) pairs in the search tree[2], and (2) the cost needed to compute the search strategy and heuristic functions on different states, in order to generate the tree itself. For instance, the time complexity is the number of CPU hours spent by the search algorithm execution, generating the search tree and visiting states, until a goal state or the cutoff is reached, while the space complexity is the maximum memory size used by the algorithm (e.g. for storing multiple states at the same time). *Search cost*

6. A *cutoff* value ($cut \in C$), which is a concrete cost value used to stop the search if the search cost exceeds this value. This value is generally used for search problems which *Cutoff*

[2]Computing the search cost would not be feasible if the search tree would contain loops, because it would not be clear how many iterations of the loop were executed.

are NP-hard (e.g. optimization problems). Examples of cutoff values include a certain number of hours or a certain number of GBs in memory usage.

Using this search model, we can map every transition – along with its input and output places – from an automated MATE attack path, in an attack net. The input and output places of a transition indicate the digital items representing the states and the search problem specification, i.e. $Spec = (s_0, suc, g, C, c)$. Note that a search problem specification can be solved in both a human-assisted and an automated way. This thesis focuses only on the automatic approaches, which – as opposed to human-assisted approaches – can be formulated as search algorithm executions, i.e. $Exec = (Spec, T, F, i, h, stg, t, cut)$, where:

- $h \in H_{ref}$ and $H_{ref} \subset H$ is the set of *reference heuristics* applicable to $Spec$ given $i \in I$, i.e. $drv(i) = H_{ref}$.
- $stg \in A_{ref}$ and $A_{ref} \subset A$ is the set of *reference algorithms* applicable given $Spec$ and h.

The set H_{ref}, and A_{ref} contain elements, which are publicly known, i.e. they have been published in the literature. For any other heuristics, information or strategies, a human-assisted task is required to develop such elements.

Best known attacker Intuitively, a search strategy and a heuristic function can be applied to many different search problem specifications. In our model this translates to the fact that there exist many possible search algorithm executions for the same search problem specification. The *best known attacker* for a given search problem specification $Spec$ is the search algorithm execution $Exec$ iff:

Best known attacker

$$\begin{aligned} \forall Exec' \neq Exec \quad &: \quad t(T) \leq t(T') \\ &\wedge \quad Exec = (Spec, T, F, i, h, stg, t) \\ &\wedge \quad Exec' = (Spec, T', F', i', h', stg', t), \end{aligned}$$

where $h, h' \in H_{ref}$ and $stg, stg' \in A_{ref}$. Intuitively, this means that: given the same way of computing the search cost, the best known attacker has the lowest search cost across all other search algorithm executions, which consist of known information, heuristics and search strategies Note that this does not include unknown information, heuristics and strategies, which could be developed via a *human-assisted MATE attack*.

Search algorithm execution Using these notions we can describe the search algorithm execution using Algorithm 1. The input of this algorithm is the search problem specification $Spec$ and the output is a state $s_g \in S$. The first steps (lines 3-4) are to initialize the search tree and the fringe F with a multiset, respectively set containing only the initial state s_0. The main body of the algorithm (lines 5-11) consists of a *while* loop which runs until one of the following conditions is true: $g(stg(F, h)) = 1$ or $t(T) \geq cut$. These are called the *stopping conditions* of the search algorithm execution, i.e. the search stops whenever the state chosen

Stopping conditions

ALGORITHM 1: $Exec$, i.e. search algorithm execution

1 **Input:** $Spec = (s_0, suc, g, C, c)$;
2 **Output:** $s_g \in S$;
3 $T = (\{(s_0, 1)\}, \emptyset)$;
4 $F = \{s_0\}$;
5 **while** *(*$g(stg(F, h)) > 0$*)* $\underline{and}$ *(*$t(T) < cut$*)* **do**
6 $s = stg(F, h)$;
7 $F = F \setminus s$;
8 $T' = T \uplus suc(s)$;
9 $F = F \cup \{s | s \in S'_s \setminus S_s \wedge T' = (S'_s, A'_s) \wedge T = (S_s, A_s)\}$;
10 $T = T'$;
11 **end**
12 **return** s_g *such that* $\forall s \in S_s : g(s_g) \leq g(s)$.

from the fringe, by the strategy is a goal state or when the search cost has reached the cutoff value. The first step inside the loop is to apply the strategy and heuristic to select (line 6) and then remove a node from the fringe (line 7). The successor function is applied to the selected state s and the result is added to the search tree (line 8); note that $\uplus$ denotes a multiset union on both search nodes and arcs. Afterwards, the search nodes added to the search tree are also added to the fringe and the search continues (lines 9-10). Finally, the algorithm returns the state s_g in the search tree, which is closest to the goal (line 12). Note that this could be a goal node if $g(stg(F, h)) = 0$, however, it could also be a non-goal node, if the cutoff caused the *while* loop to exit.

4.2.3. Estimating Search Cost

The search cost of the search algorithm execution from Algorithm 1 (i.e. $t(T)$ on line 5), consists of the total number of calls to the following functions: stg, h, suc and g. Theoretically, the search cost could be estimated without actually running the search algorithm execution, if the following values can be estimated:

- The actual cost of computing the stg, suc, h and g functions during each iteration.
- The number of *while*-loop iterations until a stopping condition is met.

Firstly, we note that all of these functions take as input one or more states. From an abstract perspective, states – in our search model – consist of classical data structures such as: sets, lists, trees or graphs. Hence, the computational cost of all of these functions will depend on some characteristics of these data structures (e.g. size of a set, average fan-in of nodes in a graph, depth of a tree, etc.). Generally, the computational cost of the four functions: stg, suc, h and g is similar for different states having the same characteristics.

Secondly, we note that the actual number of *while*-loop iterations until a stopping condition is met, depends on the order in which elements are stored in these data structures. This thesis does not claim that we are able to give a formula for computing the actual search cost based on code characteristics. Instead, this thesis focuses on estimating the average cost of a search, which also implicitly includes estimating the depth of the goal and the branching factor. For instance, the average cost of a depth-first search on a degenerate tree representing a list of hard-coded strings, depends on the size of the list of strings. The actual cost may be smaller or higher because the goal state is encountered as the first or last element of the list, respectively.

Thirdly, the search cost does not only depend on branching factor and the minimum depth of a solution in the search tree, but also on the effectiveness of the heuristic – used by the search strategy – in prioritizing the states from the fringe, to be expanded in the next iteration of the search algorithm execution. This is because heuristics have an important impact on the number of *while*-loop iterations, until a stopping condition is met. Even if the branching factor of the search tree is high, a good heuristic is able to prioritize all these branches, such that only a small number of them are explored for each node in the search tree. As we will see in Section 4.3, heuristics are the main differentiating factor between most automated MATE attacks. Most of the research literature on automated MATE attacks aims to improve the cost of previous attacks by using innovative heuristics. Therefore, it is not enough to build a model for estimating the cost of an automated attack, only from data structure characteristics and the characteristics of the search tree, we also need to factor heuristics into our model, as discussed in the following paragraph.

Characterizing heuristics Heuristics have the role of simplifying the search problem by prioritizing the choices that can be made – in each iteration of the *while*-loop from Algorithm 1 – by the search strategy. In order to do this, *heuristics* are derived via a human-assisted process using information about the search problem specification. Sources for this information are broad and include results from empirical studies that solve similar search problems, human intuition, etc.

Without the prioritization of choices, done by the heuristic, the search strategy makes an uninformed choice from all possible choices until the *goal state* is reached. Therefore, without heuristics, the characteristics of the search tree such as minimum depth of goal state and branching factor, are enough to estimate the search cost. This is actually the case for the search problem of guessing a random cryptographic key of n-bits given a (plaintext, ciphertext) pair, i.e. the total number of states, $|S| = 2^n$, gives an estimate of the number of tries that the attacker must perform in order to guarantee a successful attack. As mentioned previously, the actual number of tries may be as low as 1 if the attacker guesses the key from the first try.

In practical attacks against cryptographic cipher implementations, attackers use information sources such as cache timing side-channels [16]. Attackers formulate heuristic functions on this information in order to prune branches of the search tree, such that it is feasible

to do a brute-force enumeration on the remaining possible states (i.e. secret key values). Therefore, heuristics can be seen as a way of reducing the number of explored branches of a search tree. We can factor heuristics into our model by estimating the degree to which a certain heuristic will "reduce" the branching factor of a search tree, by guiding the search algorithm execution to visit only a subset of the possible successor states. This estimate also depends on the characteristics of the data structure contained in a state of the search tree.

Deterministic and non-deterministic search algorithm execution If a heuristic is able to perform a strictly monotonic ordering between all possible choices in all states, then the search algorithm execution is fully deterministic. Often however, heuristics are only able to reduce the possible choices to a certain set, where all states have the same (highest) utility value. At this point the best-first strategy will choose one of these "best" states uniformly at random, hence the search algorithm execution is non-deterministic. Many state-of-the-art search algorithms employ such a random choice strategy and are therefore non-deterministic. It is more difficult to predict the cost of non-deterministic search algorithm executions in comparison to deterministic search algorithm executions. Nevertheless, we can give an average estimate for it.

Average time-to-compromise Using the *Big-O* notation to characterize the worst case cost of the search algorithm execution is not in accordance with common security principles [180]. According to these principles the defender is urged to assume the best case scenario for attackers. However, using *Big-*Ω (Big-Omega) notation to characterize the best case scenario for attackers, is too vague for practitioners. Pragmatically a entity who employs obfuscation wants concrete numbers in terms of time (e.g. seconds) and computation power (e.g. a machine with 8 cores of 3.5 GHz and 64 GBs of RAM), but this is not resolved using *Big-O* or *Big-*Ω analyses. One approach along these lines is called *average time-to-compromise* [152] and it gives a quantitative estimate of the time that an IT system can withstand a cyber-attack. This idea is borrowed from physical security, where the security of safes is measured by the number of minutes it takes an attacker – having access to a certain fixed set of tools – to brake the safe [132]. We use a similar approach, however, instead of manual attacks, we measure the cost needed by automated attack steps (search algorithm executions) to reach a *goal state*, given a certain amount of computation resources, i.e. CPUs and memory.

Average time-to-compromise

4.2.4. Power of MATE Attacker

In many security settings it is common to assume that the attacker has unlimited or very high resources. For instance, in cryptographic settings, leaking the contents of a message sent between two war allies, to their common enemy (a MITM), may have insurmountable costs. Therefore, it is assumed that the MITM attacker is willing to spend a great amount of resources to decrypt messages sent between two parties (e.g. allies). However, this

assumption does not hold for several other scenarios. In this thesis we focus on software protection scenarios, where MATE attackers, often do not have such great gains.

For instance, automatic attacks on software are relevant for scenarios where a software developer employs software diversity [86, 87] via obfuscation (i.e. different end-users run differently obfuscated versions of the same program), and a MATE attacker targets all obfuscated versions of that program. This means that the amount of computing resources an attacker can spend on an automated attack must be limited in order to cope with the large amount of obfuscated versions. Moreover, automated attacks that target a large population of end-users often come in the form of PUPs [148] or *changeware* [23], executed on the machines of end-users, i.e. commodity hardware on average.

If software developers want to protect their applications (e.g. Google Chrome, Microsoft Windows, etc.), against PUPs or changeware, by using software diversity via obfuscation, then these protections must be able to withstand an automated attack with computation power equivalent to commodity hardware and not a super computer. Furthermore, the window of opportunity for such automated MATE attacks is limited (e.g. a few days, weeks, months), because of continuous software updates bring new diverse software instances containing different (combinations) of software protection, which force MATE attackers to redesign and redistribute their automated attacks. Note that, software diversity is known to cause high storage and distribution costs for software vendors who want to perform: differential updates, crash-dump debugging and binary signing by the OS vendor [23].

Another example of automated attacks on software are attacks performed by anti-virus and malware analysis engines. Similar to the previous scenario, these engines run on the machines of end-users and they must analyze binary files – which potentially are diversely obfuscated malware instances – in a matter of seconds. This means that for a malware developer, who is interested in infecting as many victims as possible, it is enough to protect his malicious software using enough obfuscation to be able to bypass an automated attack executed by an anti-virus or malware analysis engine. Even organizations performing malware analysis (e.g. McAfee, Symantec, etc.), who have more advanced hardware capabilities must process millions of different malware variants per day [148, 201]. This means they cannot invest much more than a small quota of hardware power to analyze each malicious binary.

Therefore, as opposed to encrypted communication (which must be secure against supercomputers running for a large amount of time), it is generally sufficient for software developers if diversely obfuscated applications withstand an automated attack running on commodity hardware for a limited amount of time (e.g. one hour, one month, etc.). This is true for the context of software diversity and automated MATE attacks explicitly assumed in this thesis (see Section 1.2).

4.2.5. Benefits of Search Model

Based on this model we develop a comprehensive way to characterize the effort required by the steps (i.e. transitions in the attack net), of all automated MATE attacks. In order

to characterize the strength of obfuscation transformations our frameworks enables: (1) identifying the features of the data structures of states, which have a significant influence on the cost of searching for a solution and then (2) identify how these features are affected by different obfuscation transformations. Obfuscation transformations which have the largest impact on such features and hence the effort of search, are "stronger" than other transformations, w.r.t. the corresponding automated attack. This aspect is explored in a case-study presented in Chapter 5.

The characterization of the effort required by the steps which involve search enables reasoning about the overall cost of the automated MATE attacks. This gives software developers an indication of how much time their software would resist against an automated MATE attack, because the attack will last at least as long as the search algorithm execution for one transition in an attack-net path. Knowing this software developers can employ additional software protection mechanisms – if necessary – in order to increase the time needed for executing that attack step (i.e. transition). Moreover, we can use the identified features of the data structures and a model of the strategy (optionally including a heuristic) in order to predict the time needed by the automated attack, without actually performing the search. This aspect is explored in Chapter 6.

Lastly, the data structure features and heuristics instantiated in the model, help us reason about which kinds of obfuscation transformations would increase the cost of the attack. In order to prevent or hamper paths from source to sink in an automated MATE attack, we must *apply or develop obfuscation transformations that affect these features and/or heuristics*, such that the search cost increases. Therefore, software developers can choose the strongest transformations from the point of view of resilience against all known automated MATE attacks. We will elaborate on this aspect in Chapter 7.

4.3. Survey of Automated MATE Attacks

As a support for our claim that: *all automated MATE attacks can be seen as multi-step workflows (i.e. paths from sources to sinks in an attack net), containing one or more search problems*, this section presents a survey of automated MATE attacks. We instantiate our search model from Section 4.2 for several steps of automated MATE attacks. Note that the purpose of this section is to illustrate the benefits of our model (see Section 4.2.5), on various automated MATE attacks and it is, by no means, an exhaustive survey of such attacks. For this purpose we indicate which software features are important for characterizing the effort of the attack.

Finally, different programming analysis techniques (e.g. pattern matching, taint-analysis, etc.) are presented for each attack interpretation category (i.e. syntactic and semantic) and for each attack type (i.e. code understanding, data recovery or location recovery). However, any such analysis technique is only one step of a multi-step MATE attack. Therefore, other attack types may also be achievable using the same program analysis technique within a different multi-step MATE attack.

4.3.1. Syntactic Attacks

Syntactic attacks do not interpret the program code or any artifacts (e.g. instruction trace) generated by or from it. Note that in this section we do not classify the entire multi-step MATE attack as static, instead, we refer to one particular step as being static. Such a step could very well be preceded or followed by a dynamic analysis step. The advantage of this category of attacks is that they are applicable to a broader range of programming languages and artifacts, which increases re-usability and amortizes development cost. The disadvantage is that these attacks can often be hindered by most obfuscation transformations, as will be discussed in the following paragraphs.

Data recovery via pattern matching

As the name of this particular technique indicates, the MATE attacker provides a *pattern* (e.g. a regular expression [131]), and this automated attack attempts to find all the exact matches of this pattern. The *data structure* for pattern matching is generally a list of bytes or bits (e.g. machine code, memory dumps). However pattern matching can also be performed on graphs and trees, where patterns are represented by subgraphs.

Pattern matching is also used by antivirus engines or reverse engineering tools, such as the Interactive Disassembler (IDA), where patterns are commonly called *signatures*. Such signatures or patterns represent known pieces of (malicious) code, which if identified, remove the burden of human-analysis needed to understand them. For instance, the Fast Library Identification and Recognition Technology (FLIRT) built into IDA Pro [195], recognizes library functions which have been statically linked inside of a binary executable without any debugging symbols. This process is similar to that of disassembly. However, FLIRT has a much larger database of functions, which it must recognize, compared to the relatively lower number of possible assembly instructions that need to be recognized during disassembly.

The attack-net corresponding to this automated MATE attack is illustrated in Figure 4.3. FLIRT processes the binary machine code starting from the first byte, using a buffer which

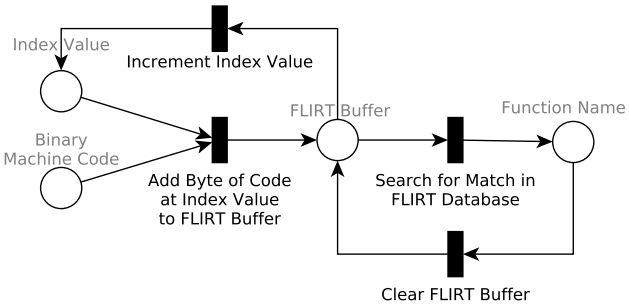


Figure 4.3.: Attack-net representing data recovery attack via pattern matching

Search Problem Specification	
Goal	A function name, whose code matches the contents of the FLIRT buffer.
Data structure	A set of sequences of machine code bytes and empty placeholders representing patterns of library functions.
State	A set of patterns and their associated function names.
Initial state	A set containing all patterns and their associated names, from the database.
Search Algorithm Execution	
Actions	Filter out patterns from current state, whose prefixes do not match the current contents of the buffer.
Strategy	Only patterns whose prefix exactly match the contents of the buffer are selected.
Heuristic	Store the set of patterns of library functions as a tree, where the root is an empty sequence and each node contains a sequence of machine code bytes and/or empty placeholders. The contents of the nodes on each path from the root to a leaf represents a function pattern. The contents of intermediate nodes are common code bytes for all function patterns of its descendants.

Table 4.3.: Elements of search specification and execution for **data recovery attack via pattern matching**.

is initially empty and performs the following steps:

1. Add the value at the current index of the machine code into the FLIRT buffer and move the index to the next byte of machine code.
2. If end of machine code is reached, then stop.
3. If the current contents of the FLIRT buffer does not match the prefix of any function templates in the FLIRT database, then empty the buffer and go to step 1.
4. If the current contents of the buffer matches the prefix of multiple function templates in the FLIRT database, then perform step 1 again.
5. If the current contents of the FLIRT buffer matches exactly one of the function templates in the FLIRT database, then assign the function name to that machine code and empty the buffer. Afterwards go to step 1.

Steps 3, 4 and 5 of the previous algorithm involve matching one of the thousands of patterns of known functions to the current contents of the buffer. This matching step maps to the transition called *Search for Match in FLIRT Database* transition in the attack-net from Figure 4.3. In the following we instantiate our search model on this transition and we reason about how to characterize its cost and by which means it could be hampered or prevented.

Table 4.3 contains the instance of our search model from Section 4.2.2 for the *Search for Match in FLIRT Database* transition in the attack-net from Figure 4.3. The *goal* is finding

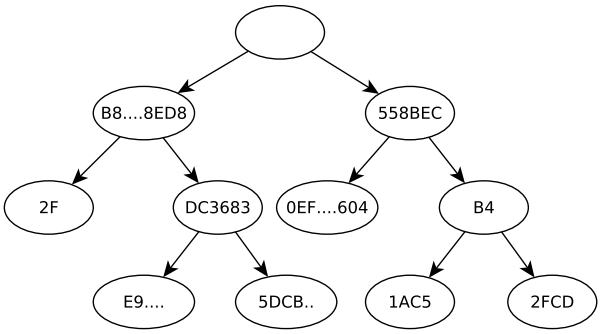


Figure 4.4.: Initial part of search tree corresponding to data recovery attack via pattern matching from Table 4.3.

a function name, whose code exactly matches the contents of the buffer, or no possible match for the buffer contents. Note that it may be necessary to search for the same function multiple times because the code is not necessarily 4 or 8 byte aligned, therefore, it is not clear at which address a function ends and another beings. The *data structure* is a set of all function patterns, which consist of machine code bytes and placeholders for bytes. The most interesting part of this attack is the *heuristic*, which is applied to the data structure, instead of the strategy, i.e. the heuristic is to store the list of function patterns inside a tree structure, which coincides with the search tree as illustrated in Figure 4.4. In this tree, functions that have matching prefixes share common ancestor nodes in the tree. Each different function pattern is reconstructed via the nodes from the root to a leaf. This heuristic reduces the effort needed for checking the content of the buffer against all possible functions in the set, to only checking those functions that have the same prefix as the current contents of the buffer. This happens because executing an action of the search by choosing a sub-tree of the search tree from Figure 4.4, is equivalent to selecting only those function patterns that correspond to that sub-tree. Differently from the previous attack, the search tree is not infinite and it is known before the search starts, because it corresponds to the database (i.e. set) of function patterns. Note that the same automated attack is also performed by code disassembly. The only difference is that instead of matching function patterns, disassembly matches instruction patterns. The *search cost* for this attack is characterized by the length of the program in bytes (denoted N), the average depth of the search tree from Figure 4.4 (denoted D), the average branching factor of the search tree (denoted B) and the average depth of the average size (in bytes) of the nodes in the tree (denoted S). If we assume that cost of comparing two bytes takes cyc CPU cycles, then the expected time for the search cost is:

$$E[t(T)] = \frac{N}{S} \cdot D \cdot B \cdot cyc.$$

Since this algorithm only involves equality comparison it is quite fast and hampering it by increasing the size of the code or the number of libraries it can recognize, will not slow it

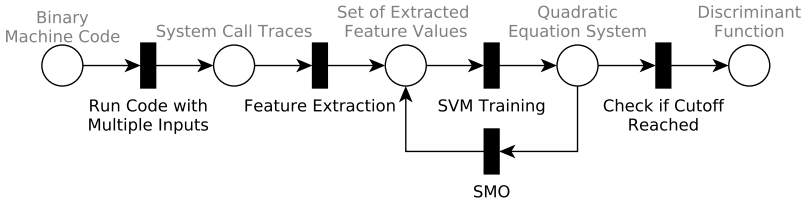


Figure 4.5.: Attack-net representing code understanding attack via pattern recognition

down much. However, preventing such exact pattern matching attacks is straightforward, because we can break the assumption that an exact match exists, by any code obfuscation technique presented in Section 3.3.2. For instance, Ibrahim and Banescu [112] successfully employ a virtualization obfuscation tool for C#, developed by Banescu et al. [20], in order to hide the location of code integrity checks against pattern matching attacks. In their case the pattern is encoded using a regular expression, which is a also a data structure heuristic, because it can store a large set of possible values in a compact finite-state automaton.

Code understanding via pattern recognition

It is not always possible to manually generate patterns for attacks. However, in the context of software diversity there are numerous instances of the same program. In *pattern recognition*, artificial intelligence algorithms are employed to automatically extract patterns from a large number of diverse software instances. The important difference is that these algorithms can potentially discover highly complex patterns, which a human attacker would not have detected.

Pattern recognition is used in behavioral malware analysis [24], which is a *code understanding* MATE attack according to our classification criteria. In this context, *behavior* is defined by the dynamic instruction trace or system call trace of a software application. Since there are many such possible traces that could be generated by malware or benign applications (i.e. goodware), supervised machine learning techniques are generally applied as part of such multi-step automated MATE attacks. Therefore, a set of dynamic traces are labeled as being generated by malware and another set of dynamic traces are labeled as originating from goodware. This way the machine learning algorithm can also eliminate patterns that are common to both malware and goodware.

Behavior

Banescu et al. [24] employ Support Vector Machines (SVMs) [61] to identify patterns in system call traces recorded during the execution of applications. The attack-net corresponding to this automated MATE attack is illustrated in Figure 4.5, where system call traces are obtained by executing the code with multiple input values. Each trace is divided using a sliding window of n system calls, where $n \in \{3, 6, 9\}$. For each window position Banescu et al. [24] extract a set of features, denoted $\vec{x_i}$, where $i \in \{1, 2, \cdots, N\}$ and N is the total number of windows from all traces in the dataset. The number of features for each window are equal to the size of the window itself, because each feature $\vec{x_i}$ represents a unique code

Search Problem Specification	
Goal	The tuple of Lagrange multipliers which solves the quadratic programming problem, representing the dual of the constraint optimization problem of finding hyperplane coefficients to discriminate malware from goodware samples.
Data structure	A tuple of Lagrange multipliers, i.e. positive real numbers.
State	A tuple of Lagrange multipliers having assigned values.
Initial state	A tuple of Lagrange multipliers with randomly assigned values.
Search Algorithm Execution	
Actions	Pick other values for one of the Lagrange multipliers.
Strategy	Sequential Minimal Optimization (SMO), which uses a divide and conquer approach, i.e. finds a pair of Lagrange multipliers at a time.
Heuristic	Select those values for one of the Lagrange multipliers in the pair, which violate the Karush-Kuhn-Tucker conditions [35].

Table 4.4.: Elements of search specification and execution for the **code understanding attack via pattern recognition**.

of a system call name. Each window position is a data point for the SVM and is labeled with a value y_i, indicating whether it is originating from a malicious ($y_i = -1$) or benign ($y_i = 1$) application's system call trace.

Discriminant function

The goal of the SVM algorithm is to identify the coefficients of a hyperplane which can discriminate malware from goodware samples (i.e. data points). A *discriminant function* tries to separate the data points such that all points labeled as malware are on one side of the hyperplane (described by this function), and all other goodware points are on the other side of the hyperplane. For instance, a discriminant function may look like $f(\vec{x_i}) = \vec{a}\vec{x_i} + b$, where $\vec{a}$ is a vector of weights and b is a constant. SVMs do not only separate points by a hyperplane, instead they use two *support vectors*, which are two hyperplanes at the boundary of each class of points. The goal of the SVM algorithm is to equivalent to finding the values of $\vec{a}$ and b, such that for all values of $\vec{x_i}$ and y_i in our dataset the following relation holds: $y_i(\vec{a}\vec{x_i} + b) \geq 1$. Support vectors are parallel to the hyperplane given by the discriminant function and these support vectors are chosen such that the distance between them is as large as possible. This is a constraint optimization problem, which can be reformulated as a quadratic programming problem solvable by searching.

Support vectors

Quadratic programming problems can be solved by a variety of methods, including the augmented Lagrangian algorithm [70], the generalized simplex method [66] and sequential minimal optimization (SMO) [171]. Banescu et al. [24] use the latter method, which takes a divide and conquer approach. Each subproblem contains one equation with two unknowns and can be solved analytically using a heuristic based search [171].

Table 4.4 presents an instance of our search model from Section 4.2.2 for this quadratic programming problem. A partial expansion of the search tree of this problem can be seen in Figure 4.6, where c and c' are two Lagrangian constants that must be determined such that

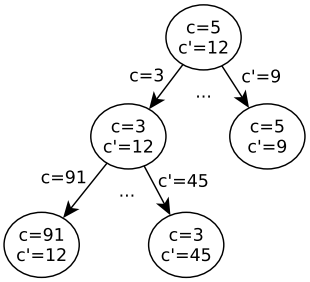


Figure 4.6.: Initial part of search tree corresponding to code understanding attack via pattern recognition from Table 4.4.

they maximize a quadratic function. Note that the branching factor and the depth of this tree are both infinite. Such problems are NP-hard and different search algorithm executions may take too much time to converge to an optimal solution, i.e. a hyperplane which perfectly partitions malware samples from goodware samples. Therefore, it is common to employ a *cutoff* time and take a suboptimal solution, e.g. Banescu et al. [24] employ a *cutoff* time of 3.5 hours in their experiments, which is also illustrated as the last transition in the attack-net from Figure 4.5. In this context, a suboptimal solution means an error in the classification, i.e. the precision and recall of the classifier are less than 100%.

These search problems are part of the *SVM Training* transition of the automated MATE attack from Figure 4.5. After training, the (suboptimal) solution is used as a model to classify other data points, as malicious or benign, based on their features. This is called the *testing phase* and it does not involve search. Therefore, we should find features which can characterize the search effort in the training phase. From the description of this automated attack, we know that the convergence time grows as a factor of the number of features n (i.e. the window size) of one data point and the number of points N in the data set. Moreover, the number of points in the dataset N, grows proportionally to the length of the system call traces used for training. If we consider that finding a pair of Lagrange multipliers using the heuristic from Table 4.4, takes cyc CPU cycles, then the expected value of the search cost is:

$$E[t(T)] = N \cdot n \cdot cyc.$$

The work of Banescu et al. [24] confirms this through their experiments which indicate that given the same cutoff time, varying the size of the window results in classification models having higher errors for higher window sizes (i.e. higher n). Moreover, using fewer traces (i.e. lower N) for training, results in models with lower classification errors.

In order to defend against such an automated MATE attack, Banescu et al. [24] proposed employing *behavior obfuscation*, which adds random noise (in the form of additional system calls), to the system call traces used for the training phase. This noise increases the data points in the dataset and also create more overlap between malware and goodware data

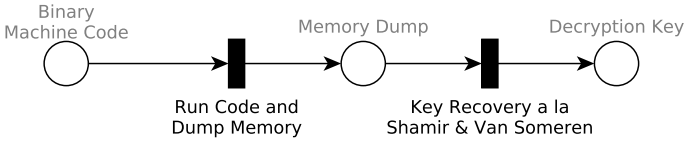


Figure 4.7.: Attack-net representing key location recovery attack via pattern recognition

points, which causes the SVM algorithm to converge slower and therefore the classification error is higher.

Location recovery via pattern recognition

Pattern recognition was also used by Shamir and Van Someren [189] to locate the secret key of an RSA cipher hard-coded in binary programs. They assume that the key is stored in memory as a contiguous sequence of v-bits chosen uniformly at random. As opposed to the previous pattern matching attack, in this attack the pattern (i.e. the actual bit values of the key), is not given. Instead, the "pattern" that is recognized is a noisy (i.e. high entropy) bit sequence which represents the key with a high likelihood, because executable code or other data values are, generally, significantly less noisy. The corresponding attack-net is depicted in Figure 4.7, where the MATE attacker is looking for a hard-coded cryptographic key of v-bits, inside of a memory dump of N-bits, where $N \gg v$. In the following we dive deeper into the analysis of the search problem in the second transition of this attack-net.

Table 4.5 presents the following necessary elements of the search model from Section 4.2.2: goal, data structure, state, initial state, actions, strategy and heuristic. The first four elements are part of the *search problem specification* and the last three are part of the *search algorithm execution*. We also depict a part of the *search tree* corresponding to this attack in Figure 4.8 (not all nodes have been expanded due to limited page width). Each node at an even depth of the search tree, has a large branching factor i.e. if the node has a sequence of N bits, the branching factor of that node is $N/2 - 1$, because one could choose any value of $x \in \{2, 3, \cdots, N/2\}$ to expand next. Since the heuristic for these nodes (see Table 4.5), assigns the same utility value to all subsequent states, the strategy picks one of these "best" states uniformly at random. Each node at an odd depth of the search tree, has a different branching factor equal to the value of $2x - 1$, where $x > 2$ is the value selected by its parent node (at the previous depth of the search tree). This happens because one can also chose two consecutive partitions, not just individual partitions, i.e. as long as not all partitions are selected. Consecutive partitions can be selected since one part of the secret key could reside at the end of one partition and the remaining part would then reside in the following partition. For instance, in Figure 4.8 at depth 1, the second child of the root ($x = 3$) has 5 children, because we also consider the states where: the first and second partitions are concatenated $P1|P2$ and the second and third partitions are concatenated $P2|P3$. For these nodes, the heuristic assigns a utility equal to the entropy of that partition,

Search Problem Specification	
Goal	The sequence of v bits representing the hard-coded cryptographic key inside a memory dump of a program P (e.g. 2048-bits for an RSA key, 128-bits for an AES key, etc.), which correctly decrypts a media file associated to P.
Data structure	A sequence of bits, representing a memory dump.
State	Range of the memory dump, i.e. start and end index of sequence.
Initial state	Range of N bits, i.e. beginning to end of the memory dump.
Search Algorithm Execution	
Actions	•For nodes (i.e. states) at even depth in the search tree (see Figure 4.8) pick an integer value of $x \in \mathbb{N}^*$, to partition range of the memory dump into x chunks of bits. •For nodes at odd depth in the search tree, pick the range of one or more chunks to partition deeper.
Strategy	Best-first search. If multiple "best" states, pick one of these states uniformly at random.
Heuristic	•For nodes at even depth in the search tree, pick $x \in \mathbb{N}^*$ greater than 1 and less than the length of the range divided by 2. •For nodes at odd depth in the search tree, pick chunks with highest entropy, because according to good security practices, RSA keys need to be randomly generated, uniformly distributed sequences of bits.

Table 4.5.: Elements of search specification and execution for **key location recovery attack via pattern recognition**.

which is a real number between 0 and 1. These values can be easily sorted and the "best" states are picked as those having the highest entropy (highlighted using a red arc label in Figure 4.8). However, note that even at odd levels in the search tree, there could be nodes that have the same entropy values and then the *best-first search* strategy picks one of these nodes uniformly at random.

Now that we have instantiated our search model, we continue with reasoning about how to characterize the attacker effort (i.e. search cost), to improve the attack and to create a defense (i.e. obfuscation transformation). The goal states are nodes in the search tree that have the range size equal to v bits. Therefore, we can infer another heuristic that: *at even levels of the search tree we must not have states with ranges lower than v bits*, because expanding such states will not lead to a goal state. The search tree is not balanced, since always picking the same value of x leads to a depth of $log_x N$, which varies for different values of x. Hence, a *depth-first search* strategy may end up exploring the deepest part of the tree, where goal states are rare. The disadvantage of breadth-first search is the large branching factor at even depths of the tree. Hence a good heuristic at even depths would dramatically reduce the cost of the attack. One heuristic that Shamir and Van Someren [189] implicitly indicate is choosing $x = v/4$. We believe that the reason for choosing this value is that it is coarse enough to indicate a meaningful entropy value and granular enough to indicate subparts

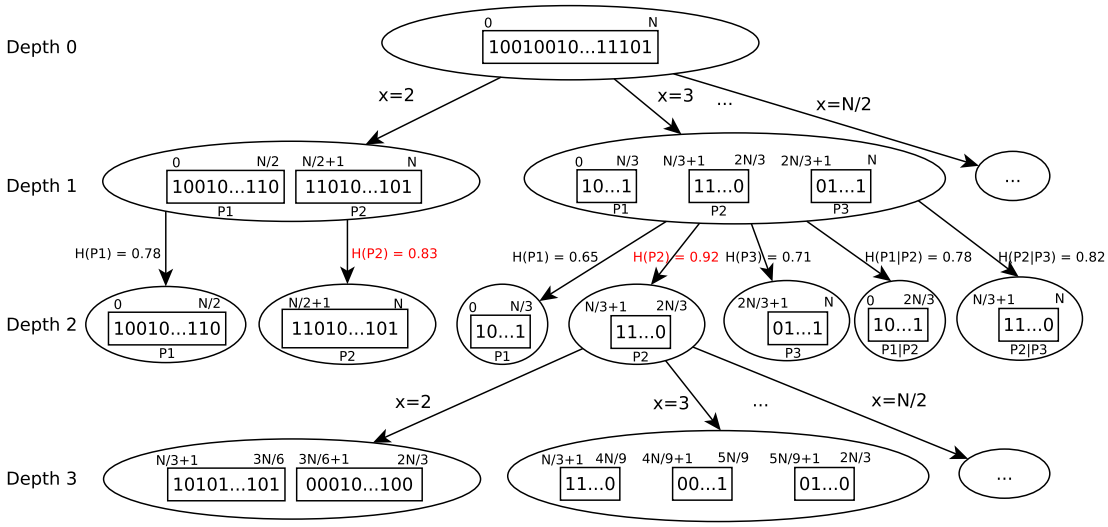


Figure 4.8.: Initial part of search tree corresponding to key location recovery attack via pattern recognition from Table 4.5. For nodes at depth 1 the best next states, have highlighted actions.

of the secret key. Using this heuristic the branching factor at even levels is reduced to 1. Moreover, the key can be found even at depth 4 in the worst case, depending on the alignment of the key inside of the memory dump. If we use cyc to denote the number of CPU cycles needed to compute the entropy of a chunk of $x = v/4$ bits, then the estimated search cost for this attack is:

$$E[t(T)] = \frac{N}{x} \cdot (cyc + \log_2 \frac{N}{x}).$$

The estimate represents the effort needed to compute the entropy of a list of N/x elements of x bits and to sort these entropy values. Afterwards, the 4 consecutive elements with the highest entropy will contain the key. Since this estimated search cost is linearithmic with the size of the program, Shamir and Van Someren [189] indicate that there is stringent need for data obfuscation methods to hide hard-coded keys in binaries. Nevertheless, the cost of this automated attack is characterized by the values of u and v, as well as the choice of x values. This attack can be stopped by breaking the last heuristic from Table 4.5, i.e. devise an obfuscation transformation which replaces the continuous sequence of bits representing the key in the memory dump, by some other data and/or code which at runtime will be used to decrypt the media file. This kind of reasoning led to the development of WBC a couple of years after this attack was published [50, 49].

4.3.2. Semantic Attacks

Semantic attacks interpret the code of the program and/or the artifacts generated by or from it. This means that moving from one state to another inside the search tree involves interpreting code. The advantage of this category of attacks is that they are more accurate than syntactic attacks. However, their disadvantage is that they are often specific to a programming language or artifact (e.g. only applicable to source code, not machine code).

Data recovery via random testing

Random testing, is a dynamic analysis technique, which randomly chooses input values for a program P and records its outputs. Therefore, it treats the program as a black-box, i.e. it does not base its decision about the next action it will take on the code of the program or any past states. Consider the automated attack of recovering a license key of variable length via random testing, which is illustrated as the top-most path in the attack-net from Figure 4.1. Table 4.6 shows the instance of this attack based on our search model. In this case the *data structure* is a string of characters and the *initial state* is the empty string. Figure 4.9 shows a partial expansion of the search tree corresponding to this search problem. Note that the branching factor of each node is equal to the size of the alphabet which is used to create the string and the depth of the tree is infinite. In Figure 4.9, the nodes are laid out in alphabetical order simply for ease of readability, however, they are actually selected uniformly at random by *random testing*. Moreover, since the program is treated as a black-box, there is no information to guide the search, therefore this attack does not use a heuristic and it uses a breadth-first search strategy.

Random testing

The cost of this automated attack depends on: (1) the depth of the goal nodes in the tree (denoted D), (2) the branching factor – equal to the size of the alphabet – (denoted S) and (3) the average number of CPU cycles needed to execute the program with a given input until the attack script can discern whether the input is a correct license key or not (denoted cyc). The expected value of the cost is therefore:

$$E[t(T)] = S^D \cdot cyc.$$

Note that in case D is known because the length of the license key is publicly known by the attacker, the structure of the search tree is different, because it can have one level with S^D values on that level. Therefore, even in this case the expected value of the search cost is the same as before. Any obfuscation transformation which slows the execution down for the license checking process would hamper this attack (e.g. garbage code insertion). It would delay the ability of the attacker to apply the goal test to the output of the program, because of the longer execution times of an obfuscated program, w.r.t. an unobfuscated program.

Location recovery via taint analysis

Up until this point we have mainly discussed passive analysis attacks, where the attacker may only observe the code of a program statically and/or dynamically. However, for several

Search Problem Specification	
Goal	An input value for the license key parameter of program P, which enables premium features.
Data structure	A string of characters.
State	A string of alphanumeric characters.
Initial state	The empty string.
Search Algorithm Execution	
Actions	Append one character from the alphabet to the string in the current state.
Strategy	Breadth-first search.
Heuristic	None.

Table 4.6.: Elements of search specification and execution for **data recovery attack via random testing**.

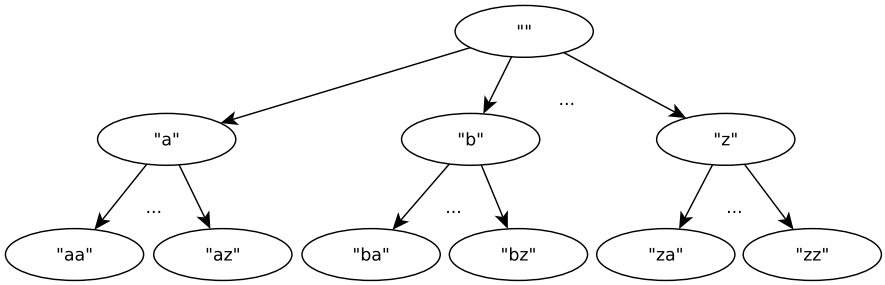


Figure 4.9.: Initial part of search tree corresponding to data recovery attack via random testing from Table 4.6.

attacker goals we must also consider active analysis attacks, where the attacker can modify the code statically or while loaded in process memory [175, 159] a.k.a. *tampering* attacks. In particular, the subgoal of identifying the location of self-checking code instructions is necessarily followed by the subgoal of disabling those instructions by modifying the code. Qiu et al. [175] propose such an attack based on taint-analysis.

Taint analysis

Taint analysis is a program analysis technique, which can be applied both to static code as well as to dynamically generated instruction traces. The purpose of taint analysis is to analyze the information flow of so called *tainted memory values*, which are chosen by the MATE attacker. These tainted memory values can be any location of memory. Popular examples of tainted memory include: user inputs, return values of system calls, file contents, etc. Taint analysis interprets code instruction-by-instruction, using an instruction pointer. It propagates the taint to other memory areas which are either directly assigned tainted values (*explicit information flow*) or whose value is affected by a conditional branch whose boolean expression is dependent on one or more tainted values (*implicit information flow*).

The automated MATE attack proposed by Qiu et al. [175], is illustrated in the attack-net

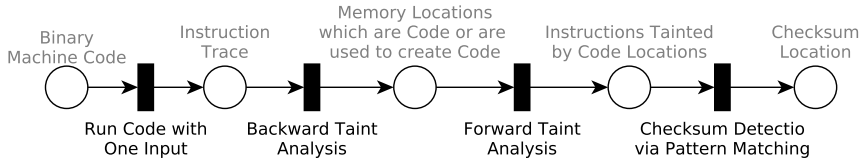


Figure 4.10.: Attack-net representing code location recovery attack via taint analysis

Search Problem Specification	
Goal	The set of all locations of code integrity checking instructions, which if patched disable code integrity protection.
Data structure	Code of program P, e.g. source code, binary code.
State	Code of P with an instruction pointer (IP) at the current instruction to be interpreted, the set of taint labels for each memory location and the set of instructions which are checks that depend on tainted memory.
Initial state	IP is set to the entry point of P and only the memory locations corresponding to the code segment are tainted, no instructions are tainted.
Search Algorithm Execution	
Actions	Interpret one of the possible next instructions according to the control-flow of the program.
Strategy	Non Uniform Random Search (NURS), i.e. select a state randomly according to a distribution given by the heuristic (below).
Heuristic	Minimum distance to uncovered instruction.

Table 4.7.: Elements of search specification and execution for **location recovery attack via taint analysis**.

from Figure 4.10. The intuition behind this attack is that if the memory area of the code segment is tainted, then code integrity checks are conditional branch instructions whose boolean expressions are dependent on tainted values. The attack requires an instruction execution trace. It applies *backward taint analysis* to this trace, i.e. it taints all instructions and traces the execution back to the memory locations which are treated as code or which are used to create code (e.g. via decryption or unpacking). After it has identified these memory locations, it taints them and performs *forward taint analysis* in order to identify the instructions, which use the tainted memory as data, not code. Consequently, it performs a pattern matching attack on the identified instructions in order to locate the code checksums.

Table 4.7 shows an instance of our search model for *Forward Taint Analysis* transition from Figure 4.10. A state for taint analysis is a set of taint labels on memory and the set of instructions which were found to be integrity checks. We describe the search algorithm execution via a concrete example. Listing 4.1 shows a C code snippet illustrating an integrity check on a code segment. Here we assume that the code segment was identified in the *backward taint analysis* step of the attack by Qiu et al. [175], and it is pointed to by the variable

Listing 4.1: Self-checksumming code example by Qiu et al. [175].

```
1 int checksum = 0;
2 for (int i = 0; i < N; i++){
3   checksum += buf[i];
4 }
5 if (checksum != V)
6   tamper_response();
```

`buf`, and has the length of `N` bytes. The checksum value is initialized on line 1 of Listing 4.1, which acts as an accumulator where the checksum of the code will reside. On lines 2-4 each byte of the code segment is added into the accumulator. Finally, line 5 checks if the dynamically computed checksum is different than a hard-coded expected value of `V`. If so, the tamper response mechanism is called on line 6. Otherwise, the code continues its normal execution.

Figure 4.11 illustrates a partial expansion of the search tree corresponding to the search problem from Table 4.7 and the example code snippet from Listing 4.1. Note that each state (i.e. node in the tree), consists of the oval where the code and instruction pointer are illustrated, as well as the rectangle to its right which contains the taint labels for each memory location and the set of instructions which are integrity checks. In the initial state, the only tainted memory values correspond to the code segment (i.e. `buf[0..N-1]`) and no integrity check instructions have been detected. After two steps the memory value of the checksum accumulator is also tainted because it is assigned values of tainted memory. One step later in the search tree, the attack identifies the first conditional branch instruction that depends on a tainted memory value. Therefore, it adds the address of this instruction to the set of integrity check instructions. Afterwards, it may continue on either the path where the condition is false or on the path where the condition is true and the tamper response mechanism is invoked. Regardless of which paths the search execution chooses to explore first, taint-analysis needs to explore all n instructions of program P, in order to determine which instructions are tainted and which not. If we denote by cyc the number of CPU cycles needed to update the taint labels in memory, then the estimated search cost of this search execution is as simple as:

$$E[t(T)] = n \cdot cyc.$$

Over-tainting

The drawback of taint analysis is *over-tainting*, which happens when memory values or instructions that are not actually dependent on tainted values are marked as tainted. For instance, we could add an opaque predicate after line 6 of Listing 4.1, which depends on the checksum and is always true. In that case, taint-analysis would mark the *if*-statement containing the opaque predicate as an integrity check instruction. Consequently, the *if*-statement would be removed along with the code inside this *if*-statement. This would break the functionality of the software after patching, by the attack.

Note that the goal of bypassing a license check may also be achieved by disabling the conditional instructions which compare the input to the license key [159]. This was indicated in the last path of the attack net in Figure 4.1. Therefore, an effective obfuscation

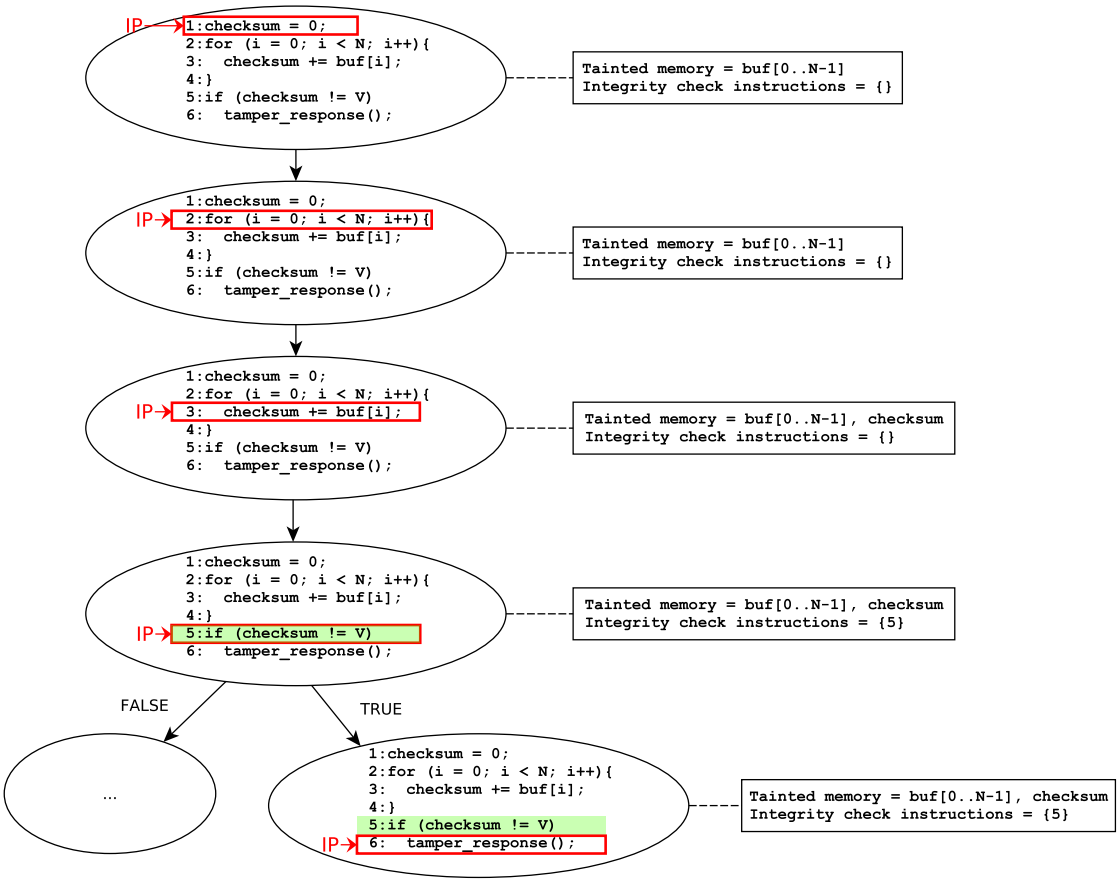


Figure 4.11.: Initial part of search tree corresponding to location recovery attack via taint analysis from Table 4.7.

transformation should not only hamper white-box test case generation, but also hamper active attacks. For instance, data obfuscation techniques such as (cryptographic) hash functions are most effective against passive attackers. However, they are easy to disable by tampering attacks [218]. Hence, we expect control-flow obfuscation techniques [59] to be more successful against active attacks. Basile et al. [30] and Varia [212] note the strong relation between obfuscation and resistance against tampering attacks. They provide formal models for tamper-resistance.

4.4. Summary

This chapter has presented the definition of an automated MATE attack as a multi-step workflow, which can be described using an attack-net. In order to characterize the effort

Attack	Description	Features for characterizing search cost	Estimated search cost
Syntactic data recovery via pattern matching	Search for matching instruction in FLIRT database	length of the program (N), average depth of search tree (D), average branching factor (B), average size of nodes (S)	$N/S \cdot D \cdot B \cdot cyc$
Syntactic code understanding via pattern recognition	Search for hyperplane which can discriminate malware from goodware samples	Number of features n and the number of points N in the data set	$N \cdot n \cdot cyc$
Syntactic location recovery via pattern recognition	Search for decryption key in memory dump of program	Length of memory dump in bytes (N), fraction of key size (x)	$N/x \cdot (cyc + \log_2 N/x)$
Semantic data recovery via random testing	Search for test case which represents license key	Depth of the goal nodes in the tree (D) and its branching factor (S)	$S^D \cdot cyc$
Semantic location recovery via taint analysis	Search for set of all locations of code integrity checking instructions	Number of lines of code (n)	$n \cdot cyc$

Table 4.8.: Summary of automated MATE attack survey.

of an automated MATE attack, a search model was proposed to describe the steps on a path from a source to a sink in the attack net. This model has been instantiated for a few automated MATE attacks proposed in the literature, however, we believe that it is possible to instantiate any automated MATE attack using our framework. A summary of the instantiations presented in this chapter, is shown in Table 4.8. Note that the first three attacks fall into the category of syntactic attacks, where code is treated as a string of bytes. The remaining two attacks described in this chapter fall into the category of semantic attacks, where code is interpreted according to its semantics. For both syntactic and semantic attacks we randomly selected three attacks – from the literature – to illustrate the three different attack types, i.e. data recovery, code understanding and location recovery. The missing code understanding attack type for the semantic interpretation category is done via symbolic execution and it will be presented in detail in Chapter 5, which is why we do not present it in this chapter.

This instantiation has facilitated reasoning about features needed to characterize the search cost of the automated MATE attacks (shown in the last two columns of Table 4.8), as well as countermeasures against these attacks. Since the next three chapters focus on automated MATE attacks based on symbolic execution, we have not discussed how to characterize such attacks in this section. The next chapter dives deeper into the problem of characterizing the effect of obfuscation on automated test case generation via symbolic execution.

5. Code Obfuscation Against Symbolic Execution Attacks

This chapter presents a characterization of automated symbolic execution attacks based on the model from Chapter 4. These characteristics are used to reason about and compare the resilience of a subset of obfuscation transformations from Chapter 3. Parts of this chapter have been published in a peer-reviewed publication [18] co-authored by the author of this thesis.

In Chapter 4 we introduced a search model that helps software developers reason about the resilience of their software against all automated MATE attacks. Instantiating this model for several steps of many different automated MATE attacks from an attack-net may seem like a daunting task. In order to remove this burden, in this chapter we first argue that there is a common step for many automated MATE attacks – even if these attacks have different attacker goals – namely generating high coverage test suites. Consequently, based on a survey state-of-the-art in test case generation by Anand et al. [2], we select symbolic execution for our case studies, because it does not require human-assistance and as opposed to black-box testing, it guides itself using the code of the program. We instantiate the search model from Chapter 4 in order to formalize the process of symbolic execution and we identify several relevant software characteristics that influence the effort of this automated MATE attack. Finally, we use these software characteristics for a systematic study of obfuscation resilience against automated attacks based on symbolic execution. This confirms that the software characteristics identified by instantiating the framework from Chapter 4, have a high impact on the effort of symbolic execution attacks.

As far as we are aware, there is no standard methodology for characterizing the resilience of different obfuscation transformations w.r.t. each other, against automated attacks. We are aware of works that focus on the empirical evaluation of the potency of obfuscation against human-assisted analysis [43, 44]. However, according to the pioneering work of Collberg et al. [58], obfuscation strength should be measured in terms of both *potency* against human-assisted attacks and *resilience* against automated attacks. This chapter is concerned with an empirical evaluation of the latter, with a focus on symbolic execution. Our work is complementary to existing work focused on human-assisted attacks, as well as works that evaluate the resilience of obfuscation against static analysis attacks [129, 65]. While focusing on symbolic execution may seem narrow in scope, it has been reported that a large number of deobfuscation techniques rely on symbolic execution [185].

5.1. A Common Subgoal of Automated MATE Attacks

Schrittwieser et al. [184] note that motivations of MATE attackers are diverse. However, their goals can be placed under the following 2 categories:

1. Extracting proprietary algorithms or data (e.g. keys, credentials) from programs.
2. Modification of software to change its behavior, also known as *software-tampering*.

These categories can be mapped to the two *code alternation* categories from Section 4.1.4, namely passive and active, respectively. In Section 4.3 we saw that the goals in the passive category can be achieved by several means, i.e. program analysis techniques. The goals in the active category can be achieved by first identifying and disabling any integrity checks on the code itself [175] and then modifying the actual functionality of the target program. We believe that there is a common subgoal for both of these goals, namely automated test case generation.

In order to identify this common subgoal we look towards the automated MATE attacks presented in Section 4.3, as well as other state of the art automatic attacks. Automatic MATE attacks are often specific to certain implementations of obfuscation transformations (e.g. virtualization obfuscation [129, 100, 190, 60], opaque predicates [65], control-flow-flattening [209]), or they are specific to certain attacker goals (e.g. CFG simplification [228], identifying code self-checks [175], bypassing license checks [21]), or both. Except for those automated MATE attacks, which only have static analysis steps in the path from source to sink in an attack-net, e.g. [189, 129], all others use dynamic analysis often in combination with some form of static analysis. The main reason for the use of dynamic analysis is that obfuscation techniques such as run-time unpacking and self-modifying code cannot be analyzed statically. However, a pre-requisite of dynamic analysis is the generation of valid inputs for the program being analyzed. How these valid inputs are obtained is not always described in works that present automatic MATE attacks. In some works they are picked randomly, in others they are generated using a symbolic execution engine.

Random test case generation is not sufficient for common attacker goals. We argue that certain automated MATE attacks (e.g. uncovering trigger-based behavior in obfuscated malware [38]), require generation of test suites that achieve up to 100% reachable code coverage. Firstly, consider the goal of simplifying the CFG of an obfuscated program as presented by Yadegari et al. [228]. In order to ensure that the CFG is complete (i.e. there are no missing statements, basic blocks or arcs), the analysis technique requires execution traces that cover all the code of the program being analyzed. Secondly, consider the goal of identifying code which verifies checksums of other parts of the code as presented by Qiu et al.[175]. To ensure that all self-checking code instructions are identified (which is mandatory in case there is a cyclic dependency between the instructions which perform checking [45, 110]), the analysis technique requires execution traces that cover all the code of the program being analyzed. Thirdly, consider the goal of bypassing a license check as presented in [21]. If the license check is performed by a conditional statement based on

an input to the program, generating a test suite that covers all the code of the program guarantees that one of the test cases contains the license key. However, unlike the previous two goals, a test suite that achieves 100% coverage of reachable code is sufficient, but not necessary, because the license key may be guessed correctly before the test suite covers 100% of the reachable code. Nevertheless, we believe that test case generation is a common prerequisite for all state of the art automatic MATE attacks. Hence, our proposal in this thesis is to characterize the resilience of obfuscation transformations based on the increase in the effort needed to generate test cases for the obfuscated program relative to its unobfuscated counterpart.

We acknowledge the fact that after achieving this subgoal, an automatic MATE attack may need to perform further tasks (i.e. transitions in the attack-net) to achieve its end goal, because this subgoal is only one transition on a path from a source to a sink in the attack-net. However, the following transitions are different for different attacker goals, while the subgoal of automated test case generation is common for most state of the art attacks. We claim that the effectiveness of an obfuscation transformation can be measured by the increase in effort (i.e. slowdown) for automated test case generation.

5.1.1. The Effect of Obfuscation on Automated Test Case Generation

The main techniques for automated test case generation according to Anand et al. [2] are: symbolic/concolic execution, model-based test case generation, combinatorial testing, fuzzing, (adaptive) random testing and search-based testing. By "automated" we mean that the user does not need to provide a list of possible input values to the test generation method, only the program source or binary code, the type and number of inputs are needed. These test case generation techniques can be further divided into *white-box testing* techniques (i.e. symbolic/concolic execution), which analyze the program code to guide test case generation and *black-box testing* techniques (i.e. model-based testing, combinatorial testing, fuzzing, adaptive random testing and search-based testing), which do not analyze but simply run the code.

Since obfuscation techniques change the code of a program but not its input-output behavior, they only affect white-box testing techniques, modulo any overhead which is also incurred by black-box testing techniques due to executing any additional instructions in the obfuscated version of a program. We hence argue that the effectiveness of applying one or more obfuscation transformations can be measured by the increase in effort needed for a white-box testing technique to generate a test suite that covers all the reachable code of the given program (e.g. for the goal of CFG simplification) or to find an input that leads to a particular execution path (e.g. for the goal of extracting a license key).

If the absolute effort for a white-box testing technique to generate a test suite for an obfuscated program surpasses the effort for a black-box testing technique to generate a test suite that covers the same paths for the same program, then the attacker will use the test suite output by the black-box testing technique. Hence, we recommend bounding the number of obfuscation transformations applied (to protect a program), by the shortest time

Listing 5.1: Program with easy to find test suite

```
1 if (x > 127)
2   // do this
3 else
4   // do that
```

needed for a black-box test case generator to produce a test suite that covers all the reachable code of a given binary or to find an input that leads to a certain execution path. For instance, consider a program with a simple control-flow structure such as the one from Listing 5.1, where `x` is an unsigned character input value. Obfuscating this program using multiple layers of obfuscation would certainly make it hard to analyze statically. However, from the point of view of dynamic analysis this program has only 2 paths. Moreover, a black-box testing technique has a 50% probability of finding a test case that covers each of the 2 paths due to the fact that the if-statement on line 1 divides the range of input values into 2 equally sized subranges. This probability changes depending on the range and number of input arguments, as well as the types of conditional branches inside the code. For instance, for a license checking function, which takes a 32 byte string as input and has only one correct license key, the probability of a black-box test case generator of finding the correct license key is $\frac{1}{256^{32}}$, which is quite low. Therefore, it is more economically attractive for a MATE attacker to analyze such a license checking function using a white-box approach.

5.1.2. Instantiating the Search Model for Symbolic Execution Attacks

Symbolic execution

Symbolic execution as originally described by King [130], involves simulating the execution of a program by replacing all input values of a program with "symbolic" values. As the simulation of the execution progresses, *path constraints* are added to "symbolic" values whenever these values are processed. When a branch condition is encountered, the simulation is forked into two paths: one path where the branch condition evaluates to true and the other where it evaluates to false. These two opposite conditions are separately appended to the path constraints existing before the branch, such that they generate two path constraints corresponding to each path.

Path constraints

Concolic or Dynamic Symbolic Execution The premise behind symbolic execution is that all code is available for simulation. However, in practice this may not hold, e.g. for system calls, which execute at OS kernel level. *Concolic execution* stands for **conc**rete + symb**olic** execution and it solves the issue of missing code by assigning concrete values to system call arguments and dynamically executing them [94, 40]. The concrete return values and side effects of system calls are then used to continue symbolic execution. Concrete value assignments are obtained by querying a Satisfiability Modulo Theories (SMT) solver using the path constraints for a certain path [89]. An SMT solver tries to find an assignment of concrete values to symbolic variables, which will satisfy all path constraints [28]. Internally

Concolic execution

Search Problem Specification	
Goal	Test suite which achieves 100% coverage of reachable program code.
Data structure	Code of program P, e.g. source code, binary code.
State	Code of P with an instruction pointer (IP) at the current instruction to be interpreted. Plus the associated memory state of P and the path constraint.
Initial state	IP is set to the entry point of P and its memory state is empty.
Search Algorithm Execution	
Actions	Interpret one of the possible next instructions according to the control-flow of the program.
Strategy	Non Uniform Random Search (NURS), i.e. select a state randomly according to a distribution given by the heuristic (below).
Heuristic	Minimum distance to uncovered instruction.

Table 5.1.: Elements of search specification and execution for **code understanding attack via symbolic execution**.

SMT solvers often convert path constraints into boolean formulas and use a Boolean satisfiability (SAT) solver to find a solution. Other analysis techniques which improve on classical symbolic execution have been developed over the last decade also under the name *dynamic symbolic execution* [39, 181, 193]. In this thesis we use the term *symbolic execution* to refer to all the techniques which employ a mix between dynamic analysis and symbolic execution.

Covering Code with Symbolic Execution

Brumley et al. [38] propose a code understanding attack based on symbolic execution, for identifying and analyzing trigger-based behavior in malware. This is motivated by the fact that malware often contains *trigger conditions*, which must be satisfied in order for malware to exhibit its malicious (interesting) behavior, i.e. if trigger conditions are not met, then the malware appears to be a benign program. Their idea for uncovering trigger conditions is to explore as many paths as possible in the code by making different possible trigger types (e.g. time, user inputs, network inputs) symbolic and keeping other variables concrete. Any branch that depends on symbolic values is potentially a trigger condition inside the malware. However, since malware may have several trigger conditions and these could occur anywhere in the program, it is necessary to cover all code and as many paths as possible, ideally all paths if their number is not infinite.

Trigger conditions

Table 5.1 shows an instantiation of our search model from Section 4.2.2, for this automated attack. The *data structure* for this search problem is the code of the program being analyzed. Additionally to this data structure the state also has the following three auxiliary digital

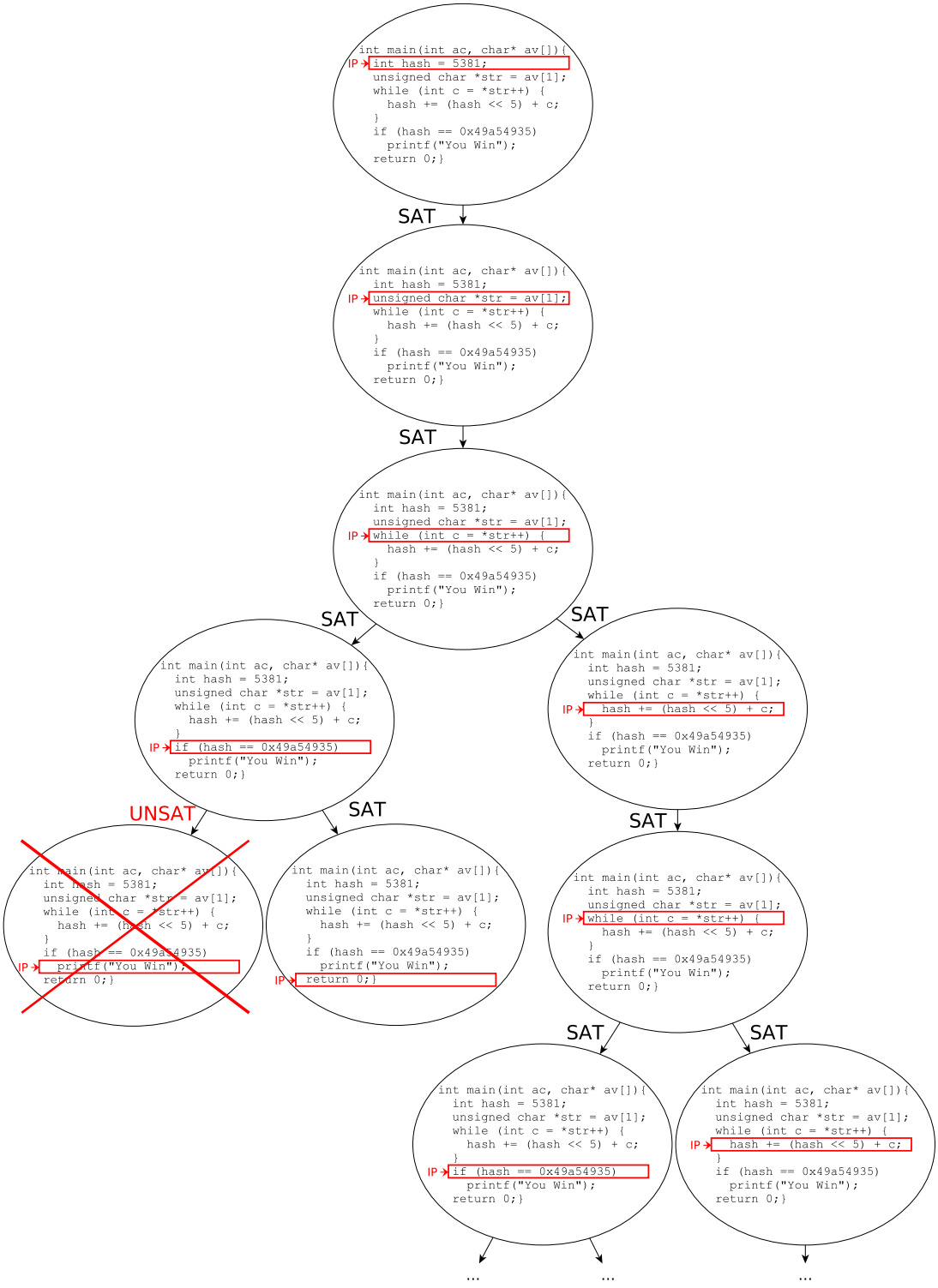


Figure 5.1.: Initial part of search tree corresponding to code understanding attack via symbolic execution from Table 5.1.

Listing 5.2: Program containing a trigger condition, i.e. it only prints "You Win" if the DBJ2 hash of the input is equal to a hard-coded value.

```
1  int main(int ac, char* av[]) {
2    int hash = 5381;
3    unsigned char *str = av[1];
4
5    while (int c = *str++) {
6      hash += (hash << 5) + c;
7    }
8
9    if (hash == 0x49a54935)
10     printf("You Win");
11
12   return 0;
13 }
```

items associated to it: (1) an instruction pointer, (2) the memory contents of the program and (3) a path constraint. Consider the simple C program from Listing 5.2. This program computes the DJB2 hash algorithm on the characters of the string passed as the first argument (lines 2-7). If the result of the hash is equal to a hard-coded value, then a the string "You Win" is printed on the standard output (lines 9-10). Otherwise, nothing is printed and finally the program stops its execution (line 12). The boolean expression from line 9 can be seen as a trigger condition for this program.

A partial expansion of the *search tree* corresponding to the symbolic execution of this program, is presented in Figure 5.1, where the instruction pointer associated with the state is denoted IP. Note that the depth of this tree is bounded by the length of the first input argument, which is unbounded if that value is symbolic. The *strategy* and *heuristic* first pick those states on the *fringe* of the *search tree* which lead to the interpretation of an uncovered instruction. As symbolic execution moves from one state to another, it records all operations performed on symbolic values as SMT formulas, i.e. *path constraints*. At each branch instruction which depends on a symbolic value, there are two possible paths corresponding to the true and false cases of the branch. Therefore, the symbolic execution engine appends the boolean condition of the branch instruction, to the current path constraints corresponding to that symbolic value and sends this as a query to an SMT solver. The symbolic execution engine also sends another query corresponding to the negation of the boolean condition. The SMT solver returns: (1) *SAT* if it finds a solution for this query, (2) *UNSAT* if the query cannot be satisfied by any possible assignment, or (3) *TIMEOUT* if it is cutoff due to the fact that a certain time limit was reached and the query could not be proved to be *SAT* or *UNSAT*. If *SAT* is returned by the SMT solver, then it has found an input to the program which can lead to that state of the program. Otherwise, if *UNSAT* or *TIMEOUT* is returned, then the corresponding state cannot be reached and it is discarded. Therefore, SMT-/SAT-solvers also perform a search algorithm execution, which affects the overall effort of the symbolic execution engine. In the next section we map this SMT/SAT search problem onto our search model, in order to identify relevant features for

characterizing the search cost.

In Figure 5.1, the left-most leaf of the search tree indicates the state where "You Win" is printed on the standard output after one iteration of the *while*-loop. Using the minimum distance to an uncovered instruction as a heuristic, the Non Uniform Random Search (NURS) strategy from Table 5.1, may try to reach this state first. This state cannot be reached, because the following query to the SMT solver has no solution:

```
5381 + (5381 << 5) + c == 0x49a54935,
```

where `c` is an unsigned character, i.e. one byte value between 0 and 255. However, this does not mean that the corresponding line of code can never be covered. The symbolic execution search strategy performs another iteration of the *while*-loop of the program from Listing 5.2 (right sub-tree in Figure 5.1), and then it again tries to enter the *if*-statement in a subsequent state. Such states are not shown in Figure 5.1 due to space limitations, however, they are children of the states that execute the *if*-statement before the print statement.

From this search model of symbolic execution presented above, we can easily identify that the search tree structure depends on the length and the number of control-flow statements of the program. The more control structures dependent on symbolic values are in the program, the more branches the search tree has. Moreover, the depth of the tree is determined by the number of iterations of the loop. On the one hand, if the loop is bounded by a large integer value, then the search tree is very deep, but the trigger condition may be satisfied before the deepest node is reached. On the other hand, if the loop is bounded by a symbolic value, then the symbolic execution engine will search for the right value such that it finds a node in the search tree where the trigger condition is satisfied. The depth of the tree could be further increased if the program had nested control-flow structures.

Solving Path Constraints via SMT-/SAT-solvers

SMT instances are a generalization of boolean satisfiability (SAT) instances by adding equality, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories [68]. SMT-/SAT-instances are also solved via search. This means that each arc of the *search tree* from Figure 5.1, involves another search performed by an SMT solver to find a concrete value for symbolic variables, which satisfies the path constraints. Since both SMT- and SAT-instances are similar, the same search algorithm can be applied to solve both of them, e.g. DPLL [67]. There are hundreds of works proposing improvements of various aspects of the DPLL algorithm published in the literature, some of the most successful are: Conflict-Driven Clause Learning (CDCL) [194] and Chaff [156]. However, all SMT-/SAT-solvers are in essence search engines, which seek a solution for any given query.

Table 5.2 shows the mapping of a random SAT problem to our search model. The *data structure* corresponding to this problem is a SAT instance (i.e. a boolean query), in conjunctive normal form, e.g.:

$$(!a + b + c) \cdot (a + c + d) \cdot (a + c + !d) \cdot (a + !c + d) \cdot (a + !c + !d) \cdot (!b + !c + d) \cdot (!a + b + c) \cdot (!a + !b + c) == 1, \quad (5.1)$$

Search Problem Specification	
Goal	A test case to cover one new path in a program, i.e. a satisfiable assignment to all literals of the SAT instance from Equation 5.1.
Data structure	SAT instance in conjunctive normal form, i.e. a set of tuples of literals, where each tuple is a disjunction of literals and tuples are in conjunction with each other.
State	A partial assignment, i.e. zero or more literals are assigned logical values. True is represented by 1 and false by 0.
Initial state	No literals are assigned.
Search Algorithm Execution	
Actions	Choose one literal or its negation to assign a logical value to.
Strategy	Non Uniform Random Search (NURS), i.e. select a state randomly according to a distribution given by the heuristic (below).
Heuristic	First assign literals that have the highest frequency in the set and tuples of the SAT instance.

Table 5.2.: Elements of search specification and execution for **data recovery attack via SAT solving**.

where a, b, c and d are Boolean variables (i.e. 0 or 1) also called *literals*, ! represents logical negation, $+$ represents logical OR and $\cdot$ represents logical AND. A *clause* is a group of literals that are OR-ed (i.e. in disjunction) to each other. Note that the previous SAT instance is a random example and it is not derived from the path constraints of the symbolic execution depicted in Figure 5.1, because such a path constraint consists of dozens of Boolean variables (corresponding to each bit of the symbolic variables in that path constraint) and hundreds of clauses, which would not be suitable as a human-readable example in this thesis. When a literal is assigned a logical 1 value, the clauses to which this literal belongs to, become true. Therefore a SAT instance is a conjunction of clauses. The goal of the SAT solver is to find a assignment to all of the literals such that the SAT instance is satisfied. In the context of symbolic execution attacks, this assignment gives the MATE attacker a concrete test case to reach a new path in the symbolically executed program.

Literals

Clause

The pseudo-code for the DPLL [67] algorithm used for solving SMT-/SAT-instances is illustrated in Algorithm 2, where *DPLL* is a recursive function which takes two arguments, namely a SMT-/SAT-instance (denoted Θ) and a constant value for a literal in Θ. The first step of the *DPLL* function is to propagate the constant value of the literal passed in as an argument, to all clauses involving this literal, which is also called *binary constant propagation*. Afterwards, *DPLL* returns *true* if all clauses in Θ are satisfied by the current assignment of literals. If one or more clauses in Θ cannot be satisfied after the binary constant propagation, then *DPLL* returns *false*. Otherwise, another unassigned Boolean variable is picked and *DPLL* is called recursively for each truth value of this Boolean variable.

Algorithm 2 is deterministic and it can be illustrated using a BDD, where each level

ALGORITHM 2: Davis–Putnam–Logemann–Loveland (DPLL)

```
1  Input: SMT-/SAT-instance, denoted Θ;
2  Output: A truth value indicating if the instance is SAT or UNSAT;
3  function DPLL(Θ, constant value for one literal ∈ Θ) {
4  Propagate constant value of literal to all clauses in Θ;
5  if satisfied: all clauses in Θ are true then
6  |  return true;
7  end
8  if conflict: one or more clauses in Θ is false then
9  |  return false;
10 end
11 for each literal ∈ Θ do
12 |  return DPLL(Θ, literal = 0) or DPLL(Θ, literal = 1);
13 end
14 }
```

corresponds to a certain Boolean variable. The algorithm simply enumerates all possible combinations of values for all Boolean variables of an SMT-/SAT-instance, and stops when a satisfying assignment is found. This is equivalent to traversing a BDD and stopping at a leaf where the assignment to variables satisfies the SMT-/SAT-instance, which is the most naïve way of searching for a solution. It is similar to enumerating all possible key values in order to guess the decryption key of a given plaintext-ciphertext pair. The complexity of this naïve way of searching is exponential in the number of Boolean variables of the given SMT-/SAT-instance, and hence not practical. However, lines 11-13 of Algorithm 2 can be slightly modified to make a more informed decision based on what was observed from previous literal assignments which caused the function to return false. The past decades have seen significant advances in the time efficiency of SAT and SMT solving, due to such changes to the *DPLL* algorithm, which employ heuristics for prioritizing the order in which literals should be assigned [89]. The state of the art algorithm which performs such a prioritization is called Conflict-Driven Clause Learning (CDCL) [194]. State of the art solvers such as MiniSAT [76] and Z3 [68] employ CDCL. They also employ a tunable degree of randomness in choosing the next literal to be assigned, because this has been observed to give better solving times in practice. Nevertheless – since the problem of SAT solving is NP-complete – in practical scenarios a *cutoff* is employed to stop the search after a certain time.

Figure 5.2 shows a partial expansion of the search tree. The branching factor depends on how many literals have not yet been assigned. In the initial state the branching factor is equal to 8, because none of the literals have been assigned. Note that assigning a value to one literal (e.g. $!a$) also assigned the opposite value to its negation (i.e. a), hence, at depth 1 in the *search tree*, the branching factor is 6, and so on. The contents of some nodes of the *search*

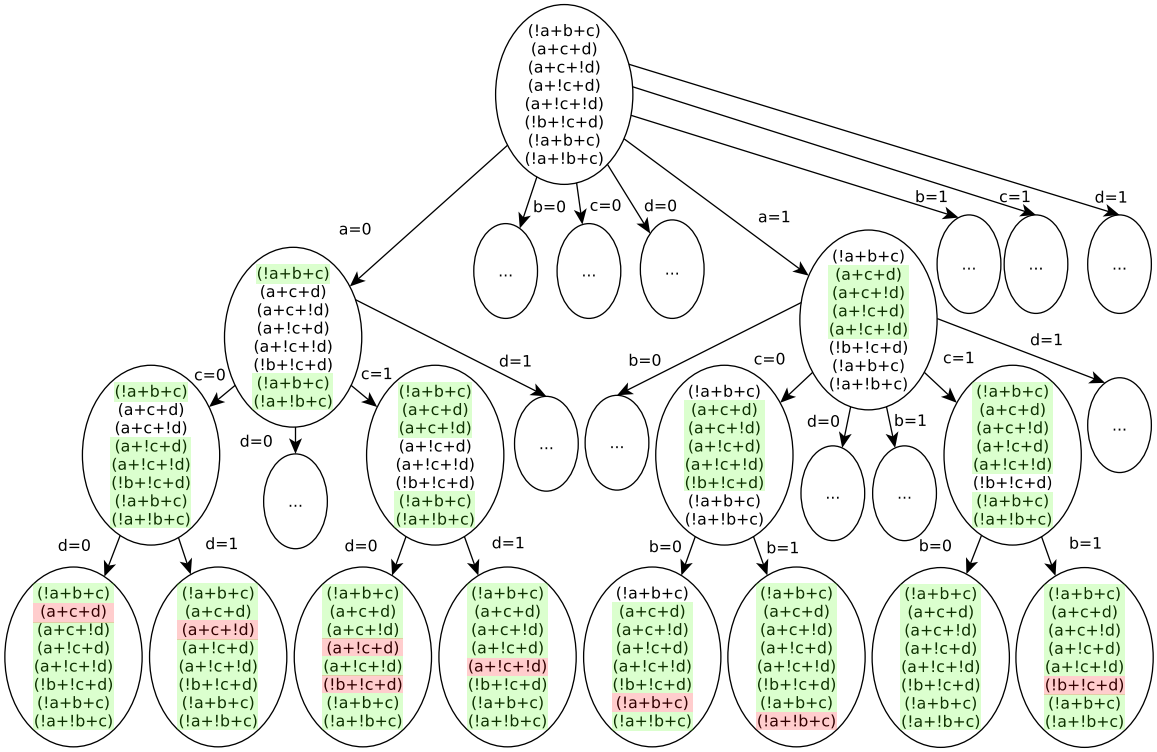


Figure 5.2.: Initial part of search tree corresponding to data recovery attack via SAT solving from Table 5.2.

tree are not described in detail due to space limitations. However, for the nodes where the state is given, we use green and red backgrounds to indicate the clauses of literals which are satisfied and unsatisfied, respectively, by any previous assignment. This (un-)satisfiability of clauses is determined by the process of Boolean constant propagation, once a literal is assigned a constant values. Note that all but one of the leaves of the search tree in Figure 5.2 contains a solution, i.e. the penultimate leaf from left to right. The assignments to reach this leaf were: $a = 1$, $c = 1$ and $!b = 1$. This also means that for this assignment both possible values for d would also lead to a solution.

From the search model of SMT-/SAT-solving presented above we can identify that the size of the search tree depends on the number of literals, which in turn depends on the size (range) of the symbolic values from the path constraints. Therefore, the data type of the symbolic variables should have an impact on the effort needed by this attack, because wider variables – in terms of number of bits – should result in SMT-/SAT-queries with more literals. Also, different types of operators in program statements (e.g. addition, division, modulo, etc.) entail different complexities of the resulting SMT-/SAT-queries. Moreover, a higher nesting level of conditional branch statements is equivalent to having a more complex boolean condition in one branch statement, which also affects the complexity

of path constraints. All of these software features will be used in the case studies from Section 5.2 and Section 6.2.

5.2. Case Study

In this section we leverage the software features that were uncovered by formalizing symbolic execution attacks as two search problems. Moreover, we measure the execution time overhead added by existing code obfuscation transformations to the symbolic execution of different programs by state-of-the-art tools. We created 2 datasets of programs[15]. Using the first dataset we analyze the symbolic execution slowdown of obfuscated programs, w.r.t. their unobfuscated counterparts, for the attacker who wishes to attain 100% reachable code coverage (Section 5.2.2), e.g. to simplify the CFG or to identify and disable self-checks. Our goal in Section 5.2.2 is to determine which obfuscation transformations from 2 freely available obfuscation tools are more resilient against symbolic execution and which transformations should be used in combination with each other. Using the second dataset of programs our goal is to compare different symbolic execution engines (Section 5.2.3) which simulate the attacker that wants to reach a certain path, e.g. to bypass a license check. Section 5.3 presents a summary and the threats to validity for this case study.

5.2.1. Obfuscator and Analysis Implementations

Commercial obfuscation tools such as Themida [205], Code Virtualizer [204], VMProtect [214] and ExeCryptor [206] operate exclusively on the Microsoft portable executable (PE) format. However, implementations of state-of-the-art test case generators for PEs such as Microsoft SAGE [96] are not publicly available. Other tools such as KLEE [39], *angr* [193] and Triton [181] have no or limited support for analyzing PEs. BitBlaze Vine [196] and S2E [48] (based on KLEE), support the analysis of PEs, however, as existing work [227] already points out, they have issues analyzing code obfuscated by the commercial tools mentioned above. Commercial obfuscators employ anti-analysis tricks (e.g. obfuscated jumps, symbolic code), which profit from the fact that the operational semantics of some of the previously mentioned symbolic execution engines do not model all details of the execution environment (e.g. the flags register). However, as shown in [227], these anti-analysis tricks can be circumvented if the symbolic execution engine semantics are updated accordingly. Hence, these anti-analysis tricks do not affect the search tree or the heuristic of symbolic execution engines as discussed in Section 5.1.2. In this work we use obfuscation transformations in the Tigress C Obfuscator [56] and Obfuscator-LLVM [120], which operate at the C source code level, respectively LLVM intermediate representation level and we do not employ anti-analysis tricks.

We used KLEE version 1.3.0 with LLVM version 3.4 and the POSIX runtime provided by its custom *klee-uclibc* version 1.0.0 and the STP SMT solver [89] version 2.2.0. Additionally to using KLEE, in Section 5.2.3 we also use angr version 4.6 and Triton version 0.3.

Code Metric	Min	Med	Avg	Max
Calculations	0	2.0	1.84	4
Conditionals	1	2.5	2.68	4
Logical	1	2.0	1.84	2
Assignment	2	3.0	3.50	6
L1.Loops	0	1.0	0.68	2
L2.Loops	0	0.0	0.11	1
Total LOC	4	13.0	12.66	18
Average.CC1.	2	3.0	2.84	3

(a) Before obfuscation

Code Metric	Min	Med	Avg	Max
Calculations	4.0	90	144.38	1343
Conditionals	1.0	37	36.36	117
Logical	1.0	10	19.88	586
Assignment	1.0	57	61.64	205
L1.Loops	0.0	1	1.40	5
L2.Loops	0.0	0	0.33	11
Total LOC	18.0	215	264.19	1881
Average.CC1.	1.5	9	11.01	50

(b) After obfuscation

Table 5.3.: Overview of manually written programs before and after obfuscation.

5.2.2. Experiment with First Dataset

The first dataset (available online) [15] contains 48 manually written C programs consisting of only one function, which prints strings on standard output after a few integer comparisons, summations and multiplications. Across all 48 programs, this function has between 4 and 18 Lines of Code (LOC) including: control-flow statements, integer arithmetic and system calls to `printf`. Note that the previous numbers of LOC do not include function declarations, comments and lines containing only closing brackets or empty spaces. Table 5.3a shows the minimum, median, average and maximum values of various code metrics of only the original (un-obfuscated) set of programs, as computed by the Unified Code Counter (UCC) tool [163]. This tool outputs a variety of code metrics including three variations of the McCabe cyclomatic complexity (CC), their average and the number of: calculations, conditional operations, assignments, logical operations, loops at different nesting levels, pointer operations, mathematical operations, logarithmic operations, trigonometric operations and total number of LOC. Each metric was computed on the main function of the C file of each program.

We have deliberately designed these programs to be small for the following 2 reasons: (1) the size of the obfuscated versions of these small programs increases between 18 and 1881 LOC (see Table 5.3b), and (2) we wanted to run a symbolic execution engine to cover 100% of the reachable code of each program and all of its obfuscated versions without any cutoff time, 10 times each, in order to check variability of the results. In case a symbolic execution engine is not able to cover 100% of the reachable code of a program, we can still perform relative overhead comparison. We can do this by simply stopping the symbolic execution of the obfuscated program when the test suite it generated has the same code coverage as the test suite generated by symbolically executing the un-obfuscated program. More generally, we must always compare the times needed to generate two test suites that have the same degree of code coverage when applied to the original or to the obfuscated program.

In Section 5.1.2 we discussed symbolic execution attacks as an instantiation of our search model from Section 4.2.2. There we mentioned a few program characteristics (i.e. length of input, structure of control flow, type of loop bounds), that would influence the

Input Size	1 byte	16 bytes
Mean	0.20	4.85
StdDev	0.52	13.54

Depth	1	2
Mean	0.05	2.63
StdDev	0.10	9.55

# Ifs	1	2
Mean	0.01	0.02
StdDev	0.00	0.01

# Input Dep. Ifs	0	1	2
Mean	0.01	0.01	0.02
StdDev	0.00	0.00	0.01

#Loops	1	2
Mean	0.03	3.63
StdDev	0.03	11.42

#Input Len. Dep. Loops	0	1	2
Mean	0.01	0.03	0.05
StdDev	0.01	0.03	0.04

#Input Dep. Loops	0	1	2
Mean	0.02	1.51	1.50
StdDev	0.02	5.90	1.42

Table 5.4.: KLEE execution time (in seconds) of original programs w.r.t. code characteristics of 1st dataset.

search tree corresponding to a symbolic execution attack (see Figure 5.1). Therefore, when creating these 48 programs we varied the following 7 code characteristics (not all possible combinations) in order to increase the heterogeneity of the programs, as well as to analyze the impact of each of these characteristics on symbolic execution. The values of these code characteristics are listed in the headers of the sub-tables from Table 5.4:

1. The size of the input of the program measured in bytes. Larger input sizes are expected to result in a larger number of literals in path constraints, but also in longer path constraints for programs which have loops bounded by symbolic values.

2. The maximum depth of nested control flow instructions (conditional branches and loops). Larger depth of control flow is expected to result in a larger branching factor for the search tree corresponding to symbolic execution.

3. The total number of if-statements. If-statements not dependent on symbolic values are not expected to increase the branching factor of the search tree corresponding to symbolic execution.

4. The number of if-statements dependent on the value of the input (as an integer). A larger number of if-statements dependent on symbolic values is expected to lead to more queries sent to the SMT-/SAT-solver.

5. The total number of loops. Loops not dependent on symbolic values are not expected to increase the branching factor of the search tree corresponding to symbolic execution.

6. The number of loops depending on the length of the input (in bytes). The longer the input the larger the depth of the search tree corresponding to symbolic execution.

7. The number of loops depending on the value of the input (as an integer). This allows variable upper bounds for loops, which is expected to result in a large depth of the search tree corresponding to symbolic execution.

Testbed description: For the experiment presented in this subsection we used KLEE to generate test suites that cover all the code in each program[1]. We have used a machine with an Ubuntu 14.04 64-bit operating system with 16GB of physical memory and an Intel Core i7-3520M CPU with 4 logical cores each having a frequency of 2.90GHz.

Step 1 - Baseline symbolic execution time before obfuscation: We ran KLEE on every original program 10 times and we analyzed the average values across these 10 runs. In this and all the of following steps (including those with angr from Section 5.2.3) the coefficient of variation c_v (i.e. standard deviation divided by the mean) of the symbolic execution time of the same program did not exceed 40% for any program. Moreover, c_v was less than 25% for over 90% of the symbolically executed programs. In other words the execution times for the same program can be considered roughly constant.

Table 5.4 shows execution time of KLEE for obtaining 100% reachable code coverage on the original (unobfuscated) programs w.r.t. the previous 7 code characteristics. It is important to note that the high standard deviation w.r.t. the mean, is computed over all programs from the dataset and denotes the heterogeneity of the dataset, which is crucial in order to have some degree of generality for our findings. The top-left sub-table in Table 5.4 indicates that the time needed for symbolic execution increases with the size of the symbolic input. The following sub-tables indicate that programs containing only if statements were faster to symbolically execute than programs which contain loops. Moreover, nesting if-statements inside loops and nested loops lead to higher symbolic execution times. Finally, making the branch condition of if-statements or loops dependent on the value of the input also increases symbolic execution time.

Step 2 - Compare symbolic execution overhead added by different obfuscation transformations: We obfuscated each of the 48 programs using the following 30 different configurations of Tigress:

1. *EncodeLiterals* (EncL): replaces constant strings and integers by code which dynamically generates these constants during execution.

2. *EncodeArithmetic* (EncA): replaces arithmetic expressions with more complex equivalent expressions.

3. *Flatten* (Flat): transforms the control-flow of a function into a flat-hierarchy of basic blocks that all have the same predecessor and successor basic blocks.

4. *Virtualize* (Virt): transforms a function into an interpreter for a random language L, translates the function's code into L and saves it as bytecode.

[1]The KLEE command used for this purpose is the following: `klee --optimize --emit-all-errors --libc=uclibc --posix-runtime --only-output-states-covering-new --max-time=3600 --write-smt2s ./filename.bc --sym-arg <input_size>`

5-8. *AddOpaque* (AddO): inserts opaque predicates i.e. branch conditions that are either always true or always false at runtime, but whose value is difficult to analyze statically. *UpdateOpaque* (UpdO): assigns new values to the variables used in opaque predicates, at runtime. We used 4 and 16 AddO, with and without UpdO.

9-28. Ordered combinations of every possible couple of the previous transformations, except for combinations with: *AddO4*, *AddO4-UpdO* and *AddO16-UpdO*.

29-30. *Flatten* and *Virtualize* were each applied 2 times consecutively on the same program.

Note that the only Tigress options used were those indicating the transformation type and the number of opaque predicates. For all other Tigress options we used the default values in order to limit the number of obfuscated programs that we then used as inputs to symbolic execution engines. The resulting code metrics as output by UCC for the obfuscated programs are shown in Table 5.3b.

Each of the 48 programs were also obfuscated using the following 9 different configurations of Obfuscator LLVM:

1. *InstructionSubstitution* (ISub): replaces arithmetic and boolean expressions with a sequence of expressions which evaluate to the same result. This transformation is similar to the *EncodeLiterals* transformation of Tigress.

2. *ControlFlowFlattening* (CFF): similar to the *Flatten* transformation of Tigress.

3. *BogusControlFlow* (BCF): similar to the *AddOpaque* transformation of Tigress.

4-9. Ordered combinations of every possible couple of the previous transformations.

All of the obfuscated programs corresponding to the first dataset were executed using KLEE. We recorded the SMT queries corresponding to every path using the `-write-smt2s` option. The value of the command line option `-sym-arg` indicates the length of the symbolic input argument passed to the program, this was changed accordingly for programs with input arguments having 1 byte and 16 bytes. We executed KLEE on every obfuscated program 10 times and we analyzed the average values across the 10 executions.

We computed the slowdown of symbolic execution of an obfuscated program w.r.t. its unobfuscated counterpart by: dividing the time needed for KLEE to analyze the obfuscated program by the time needed to analyze the corresponding original program. Figure 5.3 presents the average slowdown (using circles) in ascending order from left to right w.r.t. each employed transformations (X-axis) for the programs obfuscated using Tigress (left of vertical bar in Figure 5.3) and Obfuscator LLVM (right of vertical bar in Figure 5.3). Figure 5.3 also shows: the average increase in program size (plus signs), the percentage of time KLEE spent waiting for the SMT solver to answer all queries (solid line, only line that uses linear Y-axis on right of Figure 5.3), the average increase in total queries issued to the SMT solver (dashed line) and the average query size measured as number of nodes in the abstract syntax tree of an SMT query (dotted line). We make the following observations from Figure 5.3.

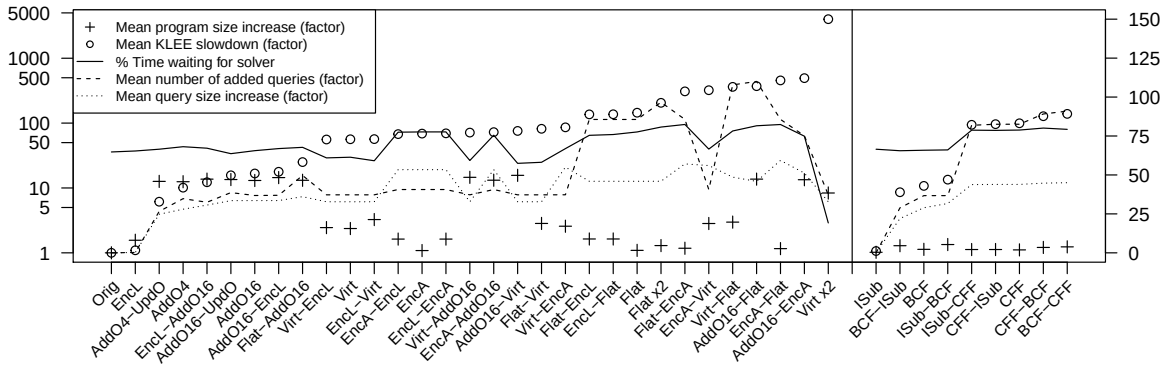


Figure 5.3.: Impact of obfuscation on the KLEE symbolic execution for programs in 1st dataset. X-axis labels to the left of the vertical bar are Tigress transformations; those to the right are Obfuscator LLVM transformations. Right Y-axis is linear and applies only to "% Time waiting for solver" (solid line). Left Y-axis is logarithmic and applies to all other curves.

Observation 1: The EncL and AddO transformations of Tigress (even when employed together or in conjunction with UpdO), have a small impact on the slowdown of symbolic execution compared to the other obfuscation transformations. The reason for this small slowdown is the fact that the additional control-flow instructions introduced by these transformations are not input dependent and they can easily be solved by the SMT solver. This observation also applies to the corresponding transformations of Obfuscator LLVM, i.e. ISub, respectively BCF.

Observation 2: EncL in conjunction with any of the 3 transformations: EncA, Flat, and Virt, does not increase the slowdown more than simply employing those transformations alone. The reason is that EncL only splits constant literals (e.g. from print statements) into a sequence of instructions, which are concretely executed by KLEE, because they do not depend on symbolic values. This observation also applies to the corresponding transformations of Obfuscator LLVM, i.e. combining ISub with CFF does not increase the slowdown more than CFF alone.

Observation 3: Resilience increases when applying Virt and Flat after any other transformation. The reason for this is that both of these 2 transformations construct an interpreter-like structure whose complexity is proportional to the size of the given source code. The other 3 transformations of Tigress tend to add more code to the input program, hence applying them before Virt and Flat results in larger interpreter-like structures. This observation also applies to the CFF transformation in Obfuscator LLVM. Moreover, applying Flat after *Virtualize* increases the slowdown.

Observation 4: The average percentage of time spent waiting for the SMT solver to solve path constraints in the original program is equal to 66.35% for the original programs in the first dataset. EncL, AddO, ISub and BCF do not have a significant impact on this value.

This is because very few additional queries are added by EncL and ISub. Also KLEE can simplify the majority of path constraints added by AddO and BCF via caching query results, i.e. without calling the SMT solver for each path constraint. On the other hand, EncA, Flat, and CFF each cause an increase of about 10% when used separately. EncA and Flat cause an increase of about 15% when used together. However, applying Flat twice does not lead to a 20% increase. Contrarily, Virt decreases the time spent waiting for the SMT solver by about 5% when used alone (despite issuing more and larger queries) and by 5-45% when used in combination with EncL, AddO and Virt.

Observation 5: Flat and CFF on average lead to an increase of 2 orders of magnitude in the number of queries issued by KLEE to the SMT solver. The queries are also approximately 10 times larger than the original queries. This indicates that the slowdown is due to the fact that KLEE is issuing many more large queries to the SMT solver which is expensive in terms of time. These queries are generated due to branches which are dependent on program input values. Programs which do not contain loops or if statements dependent on input values have no increase in the number of issued queries. An important advantage of flattening is that it has the smallest effect on program size compared to all other employed transformations.

Observation 6: EncA increases the query size 20 times. This suggests that the slowdown for this technique is due to the difficulty of solving the queries and not to the large number of queries as was the case in *Observation 5*. These queries are more difficult to solve due to the non-trivial complexity of the arithmetic expressions added by EncA. Remarkably, the size of the program is affected less by this transformation than by Virt, AddO and even EncL, because the complex expressions added by EncA are not as large as control flow statements added by the other transformations.

Observation 7: Virt tends to reduce the time that KLEE waits for the SMT solver. The reason for this reduction is due to the fact that this transformation adds more instructions to the execution paths, i.e. loading bytecode values and data values for every original statement of the program. Since all instructions have to be interpreted by the symbolic execution engine, this translates into higher slowdown. Applying Virt multiple times yields multiple levels of interpreter *fetch-decode-dispatch* instructions, hence the large slowdown when virtualization is applied twice. Similarly to flattening, Virt induces a higher slowdown on programs with loops or if-statements depending on program inputs.

Step 3 - Compare SMT instances of original and obfuscated programs: KLEE outputs an SMT file for every path that it finds in the program it analyzes. These SMT instances contain the path constraints necessary to generate a test case that leads to the corresponding path. We compared the SMT instances output by KLEE for all programs in the dataset (including the original programs).

Observation 8: The sets of SMT instances corresponding to each obfuscated version of a program are practically identical to the set of SMT instances of the original (unobfuscated) program, except for EncA, which were significantly larger. For all transformations (except

`RandomFunsOperators` Parameter Value	Description
`PlusA, MinusA, Lt, Gt, Le, Ge, Eq, Ne`	Simple arithmetic and comparison operators
`PlusA, MinusA, Mult, Div, Mod, Lt, Gt, Le, Ge, Eq, Ne`	Harder arithmetic and comparison operators
`Shiftlt, Shiftrt, BAnd, BXor, BOr, Lt, Gt, Le, Ge, Eq, Ne`	Shift, bitwise and comparison operators
`PlusA, MinusA, Mult, Div, Mod, Lt, Gt, Le, Ge, Eq, Ne, Shiftlt, Shiftrt, BAnd, BXor, BOr`	Harder arithmetic, shift, bitwise and comparison operators

Table 5.5.: *Operator* parameter values given to C code generator used for generating dataset.

EncA) only small differences occur such as using shifting and adding instead of multiplication. This is due to the fact that KLEE uses concrete values for instructions added by the majority of obfuscation transformations employed in this case-study. The concrete values can be simplified away from path constraints even without calling the SMT solver. This indicates that simply adding any additional computations is not enough to have a significant impact on the path constraints, hence the queries sent to the SMT solver. In order to have a significant impact on such queries the obfuscation transformation must add computations involving variables which will be marked as symbolic and these computations must not be trivial to solve via static techniques such as constant folding, common subexpression elimination, etc.

Observation 9: The time the symbolic engine waits for the SMT solver to find an answer to the query accounts for almost all the increase in effort added by obfuscating a program using EncA. Therefore, one possibility for improving obfuscation is to make these expressions more complex and difficult to solve by applying the EncA transformation multiple times to the same program. However, such an obfuscation could be bypassed by an attacker who knows all the substitution rules that the EncA transformation can use and then simply applies them backwards. Rolles [177] has developed a method that could automatically extract such substitution rules from programs by employing SMT solvers. In general, the complexity of SMT instances can be increased by non-linear transformations such as those employed in cryptographic hash functions (see Section 7.2).

Observation 10: None of the obfuscation transformations insert additional paths dependent on input values to the program, i.e. the sizes of the sets of SMT instances corresponding to each obfuscated version of a program have the same size as the set of SMT instances of the original (unobfuscated) program. This may be counter intuitive since AddO and BCF insert additional if statements. Since these if statements are not input dependent and always have the same truth value, the SMT solver is able to eliminate the dead branches of these if-statements, i.e. the symbolic execution engine will not analyze the code in those dead branches.

`RandomFunsControlStructures` Parameter Value	Control-flow depth	Number of *if*-statements	Number of Loops
(if (bb n) (bb n))	1	1	0
(if (bb n))(if (bb n))	1	2	0
(if (bb n))(if (bb n))(if (bb n))	1	3	0
(if (if (bb n) (bb n)) (bb n))	2	2	0
(if (if (bb n) (bb n)) (if (bb n) (bb n)))	2	3	0
(if (if (if (bb n) (bb n)) (bb n)) (bb n))	3	3	0
(if (if (if (bb n) (bb n)) (if (bb n) (bb n))) (bb n))	3	4	0
(if (if (if (bb n) (bb n)) (if (bb n) (bb n))) (if (bb n) (bb n)))	3	5	0
(for (bb n))	1	0	1
(for (if (bb n) (bb n)))	2	1	1
(for (bb n))(for (bb n))	1	0	2
(for (for (bb n)))	2	0	2
(for (if (if (bb n) (bb n)) (bb n)))	3	2	1
(for (if (bb n) (bb n))(if (bb n) (bb n)))	2	2	1
(for (if (if (bb n) (bb n)) (if (bb n) (bb n))))	3	3	1
(for (for (if (bb n) (bb n))))	3	1	2

Table 5.6.: *Control structure* parameter values given to C code generator used for generating dataset.

5.2.3. Experiment with Second Dataset

The second dataset contains 4608 C programs consisting of a main function and another function f randomly generated by the *RandomFuns* feature of Tigress. To generate the programs in this dataset we have also leveraged the program features identified when mapping symbolic execution to our search model, in Section 5.1.2. A set of parameters of the *RandomFuns* feature of Tigress, were added – by the author of Tigress – in order to cater to the code features we have identified in Section 5.1.2. The following is a list of parameters and their corresponding values we used to generate this dataset:

- The random seed value: `Seed` $\in$ {1,2,4} (3 values). This parameter has no effect on symbolic execution time. However, we use it in order to generate more than one program instance for the same values of all other parameters.
- The data type of variables: `RandomFunsTypes` $\in$ {char, short, int, long} (4 values). This parameter affects the complexity of path constraints because larger data types for symbolic values leads to SMT-/SAT-instance with a larger number of literals.
- The bounds of *for*-loops: `RandomFunsForBound` $\in$ {constant, input, boundedInput} (3 values). This parameter affects the depth of the search tree corresponding to symbolic execution, because in includes variable upper bounds for loops.
- The operators allowed in expressions: `RandomFunsOperators` presented in Table 5.5 (4 values), which also describes each parameter value. This parameter affects the complexity of path constraints, because different operator types impose various degrees of difficulty for SMT-/SAT-solvers.
- The control structures: `RandomFunsControlStructures` presented in Table 5.6 (16 values), which also shows the depth of the control flow. The grammar for this

Code Metric	Min	Med	Avg	Max
Calculations	10.00	27.00	34.64	152.00
Conditionals	7.00	10.00	10.02	16.00
Logical	4.00	9.00	12.17	69.00
Assignment	9.00	17.00	18.13	46.00
L1.Loops	2.00	3.00	2.85	4.00
L2.Loops	0.00	0.00	0.19	1.00
Total LOC	32.00	66.00	78.00	288.00
Average CC	2.67	3.33	3.21	4.00

(a) Before obfuscation

Code Metric	Min	Med	Avg	Max
Calculations	22.00	98.00	183.36	870.00
Conditionals	4.00	21.00	105.41	504.00
Logical	4.00	14.00	63.75	458.00
Assignment	10.00	32.00	222.88	1078.00
L1.Loops	2.00	3.00	2.99	10.00
L2.Loops	0.00	0.00	0.25	12.00
Total LOC	42.00	168.00	578.64	2932.00
Average CC	1.80	5.25	15.73	66.75

(b) After obfuscation

Table 5.7.: Overview of randomly generated programs.

parameter allows specifying a nested control-flow structure consisting of if and for statements. Opening a parenthesis increases the nesting level of control flow. The grammar also allows specifying the size of the basic blocks (denoted bb), via an integer number n. This parameter influences the number of paths, hence queries sent to the SMT-/SAT-solver.

- The number of statements per basic block was changed via the value of $n \in \{1, 2\}$ from Table 5.6. This parameter affects the length of the search tree for symbolic execution as well as the length of path constraints.

The total number of combinations is therefore: $3 \times 4 \times 3 \times 4 \times 16 \times 2 = 4608$. All other parameters were kept to their default values, except for the `RandomFunsPointTest`, which was set to true, meaning that the return value of the randomly generated function is checked against a constant value and if they are equal the program prints a distinctive message, i.e. "You win!" to standard output. We have set this constant value to be equal to the output of the randomly generated function when its input is equal to "12345". Therefore, all of the 4608 programs will print "You win!" on the standard output if their input argument is "12345".

Table 5.7a shows the minimum, median, average and maximum values of various code metrics of only the original (un-obfuscated) set of programs, as computed by the UCC tool [163] and the total number of LOC. Each metric was computed on the entire C file of each program, which includes the randomly generated function and the main function.

Each generated function takes an array of primitive type (e.g. char, int) as input (i.e. `in`) and outputs another array of primitive type (i.e. `out`), as shown in Listing 5.3. Each function first expands the input array into a (typically larger) state array via a sequence of assignment statements containing operations (e.g. arithmetic, bitwise, etc.) involving the inputs (lines 3-5). After input expansion, the values in the state array are processed via control flow statements containing various operations on the state variables (lines 6-17). Finally, the state array is compressed into the (typically smaller) output array via assignment statements (lines 18-19). These three phases represent a generic way to map data from an input domain to an output domain, as a license check would do. The *if*-statement on lines

Listing 5.3: Randomly generated program example.

```
1  void f(int *in, int *out) {
2    long s[2], local1 = 0;
3    // Expansion phase
4    s[0] = in[0] + 762;
5    s[1] = in[0] | (9 << (s[0] % 16 | 1));
6    // Mixing phase
7    while (local1 < 2) {
8      s[1] |= (s[0] & 15) << 3;
9      s[(local1 + 1) % 2] = s[local1];
10     local1 += 1;
11   }
12   if (s[0] > s[1]) {
13     s[0] |= (s[1] & 31) << 3;
14   } else {
15     s[1] |= (s[0] & 15) << 3;
16   }
17   s[0] = s[1];
18   // Compression phase
19   out[0] = (s[0] << (s[1] % 8 | 1));
20 }
21 void main(int ac, char* av[]) {
22   int out;
23   f(av[1], &out);
24   if (out == 0xa199abd8)
25     printf("You win!");
26 }
```

24-25 resembles a license check, where the output of the randomly generated function `f` is compared against a hard-coded value. Note that the program illustrated in Listing 5.3 is an overly simplified instance of the programs generated by the Tigress. The programs in our dataset are larger and they also contain more complex control-flow structures and boolean conditions (including disjunctions and conjunctions) in control-flow statements. Finding an input value that passes this comparison is harder for a white-box test case generator to find, than an input that would fail the comparison. Hence, this dataset resembles license checking mechanisms, which would be part of larger programs such as games or professional editing and design software, etc. Note that programs containing license checking mechanisms are much larger than our randomly generated programs, however, an attacker would not symbolically execute the entire program. Instead, an attacker would isolate the license checking code and then proceed to symbolically execute only this fraction of the program.

Testbed description: For the experiment described in this subsection we used a machine with more cores to enable running multiple symbolic executions in parallel. The machine uses the Ubuntu 14.04 64-bit operating system and it has an Intel Xeon E5-1650v2 CPU with 12 logical cores each running at 3.50GHz and 64GB of physical memory.

Step 1 - Baseline symbolic execution time before obfuscation: The impact of these code characteristics on the execution time of KLEE given the original (unobfuscated) programs

Data Types	char	short	int	long
Mean	1.32	9.95	13.41	13.91
StdDev	0.98	6.48	7.86	8.34

Loop Bound	Constant	Bounded Input	Input
Mean	8.45	8.43	10.62
StdDev	8.07	7.28	9.65

Operators	Bitwise	Simple Arith.	Harder Arith.	All
Mean	4.74	8.91	9.97	11.23
StdDev	5.60	6.81	8.60	8.60

Depth	1	2	3
Mean	7.75	9.35	9.96
StdDev	6.22	8.00	8.90

Total # Ifs	0	1	2	3	4	5
Mean	6.34	7.74	7.94	11.08	12.4	14.87
StdDev	4.92	5.90	6.51	9.22	9.35	11.97

Total # Loops	0	1	2
Mean	12.27	6.34	6.4
StdDev	9.64	4.57	5.11

Table 5.8.: KLEE execution time (seconds) on original programs w.r.t. code characteristics of 2nd dataset.

can be seen in Table 5.8. As was the case with the input size in Table 5.4, the symbolic execution time increases with the size (ranges) of the data types. This is due to the fact that for higher ranges of values it is more difficult for SAT solvers to find a solution to queries derived from path constraints on symbolic values. Different types of bound conditions placed on loop statements cause a mild difference on symbolic execution time, i.e. if the loop iterates a constant number of times, then the symbolic execution engine will execute faster on average than if the number of loop iterations depends on the program input. Finally, the type of operators used by the program has an important impact on symbolic execution, because these operators are used by path constraints which are issued as queries to the SMT solver. We notice that bitwise operators are easier to solve than arithmetic operators. The type of arithmetic operators does not cause a large difference in symbolic execution. However, harder arithmetic tends to be slower to solve than simple arithmetic. More importantly, combining all operators seems to have an additive effect w.r.t. the time taken to solve path constraints.

Step 2 - Compare symbolic execution overhead of different tools after obfuscation: We have obfuscated the f functions with 5 obfuscation transformations from the Tigress tool: AddO, EncA, EncL, Flat and Virt. We only chose these 5 transformations, due to the fact that Obfuscator LLVM transformations are very similar to AddO, EncL and Flat. Table 5.7b shows the minimum, median, average and maximum values of various code metrics of only the obfuscated set of programs, as computed by the UCC tool [163], and the total number of LOC. Note that the majority of programs have now increased their LOC by one order of magnitude.

For this experiment we used KLEE and *angr* as symbolic execution engines and we let them run until they found the path in the program that prints a distinctive message on the standard output or the timeout of 1 hour is reached. When this path is entered we know that the check guarding that path has been bypassed by the symbolic execution engine. We did not use angr in the previous experiment because angr does not aim to achieve 100% code coverage, as opposed to KLEE. Note that we have also tried to employ the Triton symbolic execution engine [181] on the obfuscated programs for both datasets. However,

	KLEE			angr		
	Median	Mean	StdDev	Median	Mean	StdDev
AddO16	0.97	1.03	0.26	1.72	2.25	2.49
EncA	1.14	1.21	0.37	1.39	1.79	1.90
EncL	0.98	0.99	0.22	1.40	2.22	4.60
Flat	1.15	1.22	0.44	3.77	4.45	2.85
Virt	1.53	2.08	1.27	7.32	8.85	5.01

Table 5.9.: Symbolic execution slowdown on programs obfuscated using Tigress, relative to unobfuscated counterparts from 2nd dataset.

Triton crashed when symbolically executing programs obfuscated using Flat and Virt due to insufficient memory. Triton transforms each assembly instruction into a sequence of SMT constraints, which increases directly proportional to the execution trace, which is large for programs obfuscated with Flat and Virt.

Table 5.9 shows the median, mean and standard deviation of symbolic execution slowdown on programs obfuscated from the second dataset w.r.t. their unobfuscated counterparts. The slowdown is computed as the time needed to symbolically execute an obfuscated program until the path in the program that prints the distinctive message on the standard output is found, divided by the time need to symbolically execute the unobfuscated version of the program to find the corresponding path. The median and standard deviation were taken across 12713 obfuscated programs successfully analyzed by the KLEE and angr within the 1 hour time limit. We make the following observations using Table 5.9.

Observation 11: KLEE incurs a lower slowdown than angr for all of the 5 obfuscation transformations employed in this experiment. This indicates that KLEE is the *best known attacker* for the obfuscation transformations we have employed in this chapter. Note that KLEE also has limitations, e.g. it does not support `goto` instructions or in-line assembly in C programs. However we see this as a technical, not a fundamental limitation.

Observation 12: The slowdown of finding the path that prints a distinctive message ("win") is much lower than the slowdown for covering all reachable code (which was the goal of the attacker in Section 5.2.2). This is expected since the symbolic execution engine may discover that particular path before covering all reachable code.

5.3. Summary and Threats to Validity

In Section 5.2.2, we have symbolically executed a set of programs obfuscated with 39 different configurations of 8 transformations from 2 obfuscation tools. We generated 100% reachable code coverage test suites that would be used by an attacker who aims to simplify the CFG of an obfuscated program. The results indicate that all the considered obfuscation transformations can be broken with different computational effort, which also depends on code characteristics of the original program. EncL and ISub are not effective against symbolic execution. AddO and BCF have a mild effect on the slowdown. EncA slows down symbolic execution due to the larger size of SMT queries. Flat and CFF slow down

symbolic execution due to the larger number of SMT queries issued to the solver. Virt slows down symbolic execution due to the high number of fetch-decode-dispatch instructions added. The results also indicate in which order obfuscation transformations should be applied to increase effectiveness, i.e. in general it is better to first add bogus code via opaque predicates, instruction substitution or by transforming constant literals into code and then applying control-flow and arithmetic obfuscation via Virt, Flat/CFF and EncA, respectively. However, we note that none of these obfuscation transformations add input-dependent paths to the programs. Therefore, in Chapter 7 we propose transformations which do insert input-dependent paths and measure how they affect symbolic execution time.

In Section 5.2.3 we have symbolically executed another set of programs obfuscated with 5 representative transformations from Tigress used in Section 5.2.2. Transformations from Obfuscator LLVM were not used in Section 5.2.3 because they had similar effects as their corresponding transformations from Tigress. In this second experiment (from Section 5.2.3), we compared the performance of different symbolic execution engines to find test cases that lead to a certain (difficult to reach) path in the obfuscated programs. This goal is different from obtaining 100% code coverage, which was the goal of the experiment from Section 5.2.2, because it represents an attacker who wants to bypass a license check in a program. Such an attacker does not necessarily need to generate a test suite that covers 100% of the code, in order to obtain the correct license key. We observed that for this goal, KLEE is most effective, followed by angr. However, the overhead of successfully recovering the correct license key via symbolic execution is about one order of magnitude smaller than that of covering 100% of the code. This indicates that protecting a license-checking mechanism against symbolic execution, requires a combination of more obfuscation transformations, than it would be necessary to protect against an attacker who aims to explore 100% of the code.

The results from Section 5.2.2 and Section 5.2.3 do not generalize for all possible programs. However, we believe that they have some degree of generality due to our carefully constructed datasets and the intuitive explanations we provide in our observations above. Note that the observations regarding program characteristics which influence the attack time (e.g. observation 3), are facilitated by the fact that we mapped symbolic execution and SAT solving to our search model in Chapter 4. Moreover, for the attacker goal of bypassing license checks – of the experiment from Section 5.2.3 – it is not feasible to simply take random programs from open source repositories (e.g. Github), because not all programs contain such checks. Artificially inserting checks into such programs would lead to the same threat to external validity as mentioned in this paragraph.

Symbolic execution has several limitations when applied to large software programs [3, 185]. However, we note that symbolic execution is currently a highly active field of research and has been successfully applied for finding bugs in Microsoft Windows 7 [95] and it is being used by several teams in the DARPA Cyber Grand Challenge for automated exploit generation [11]. Moreover, several deobfuscation techniques rely on symbolic execution [185]. Also, in practice, a software developer will not obfuscate all the code of a program, but only sensitive parts of it (e.g. license checking mechanisms). Furthermore,

obfuscation is often applied only to parts of software which are not frequently executed (i.e. so called "hot code"), in order to minimize performance impact [57], hence attackers may isolate and symbolically analyze smaller parts of the software [3].

6. Predicting Cost of Symbolic Execution Attacks on Obfuscated Code

This chapter presents a framework for predicting the time required by a successful symbolic execution attack, on obfuscated programs. The framework requires the metrics stated in Chapter 5. Parts of this chapter have appeared in a publication [19] co-authored by the author of this thesis.

In Chapter 5 we saw that symbolic execution attacks are able to successfully analyze programs obfuscated using a set of different data- and code-transformations. We have both measured and compared the time needed for such an automated MATE attack when applied to programs protected by different combinations of obfuscation transformations. However, estimating the time an obfuscated program is able to withstand a given automated MATE attack is still an open challenge for software obfuscation.

This chapter proposes a general framework for choosing the most relevant software features to estimate the effort of automated attacks. Our framework uses these software features to build regression models that can predict the resilience of different software protection transformations against automated attacks.

Resilience

Resilience is defined as a function of *deobfuscator effort*[1] and *programmer effort* (i.e. the time spent building the deobfuscator) [58]. Nonetheless, in many cases we can consider the effort needed to build the deobfuscator to be negligible, because an attacker needs to invest the effort to build a deobfuscator only once and can then reuse it or share it with others. There have been works that propose measures for resilience. Udupa et al. [209] propose using the edit distance between control flow graphs of the original code and deobfuscated code. Mohsen and Pinto [155] propose using Kolmogorov complexity. Banescu et al. [21] propose using the effort needed to run a deobfuscation attack. However, none of these works attempt to predict the effort needed for deobfuscation, which has been identified as a gap in this field [199]. In this chapter we focus on predicting the effort needed by a deobfuscation attack.

[1] In this thesis we quantify deobfuscator effort via the time it takes to run a successful automated MATE attack on a certain hardware platform; however, note that we could easily map time to CPU cycles, to provide a hardware independent measure of attack effort.

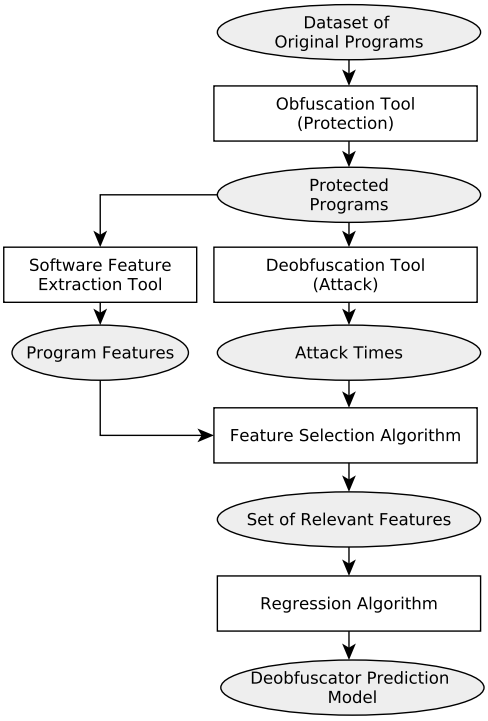


Figure 6.1.: General attack time prediction framework.

6.1. A General Framework for Predicting the Cost of Automated MATE Attacks

Our general approach is illustrated as a work-flow in Figure 6.1, where ovals depict inputs and outputs of the software tools, which are represented by rectangles. The work-flow in Figure 6.1 requires a dataset of original (un-obfuscated) programs to be able to start. Afterwards, an obfuscation tool is then used to generate multiple obfuscated (protected) versions of each of the original programs. Subsequently, an implementation of a deobfuscation attack (e.g. control-flow simplification [228], secret extraction [21], etc.) is executed on all of the obfuscated programs, and the time needed to successfully complete the attack for each of the obfuscated programs is recorded. In parallel, feature values are extracted from the obfuscated programs. These features are identified by first mapping the automated MATE attack to our search model presented in Section 4.2.2, and then reasoning about the characteristics that affect the search algorithm execution elements, namely search tree, fringe, heuristic and search strategy.

Once the *attack times* are recorded and software features are extracted from all programs, one could directly use this information to build a regression model for predicting the time needed for deobfuscation. However, some features could be irrelevant to the deobfuscation

attack and/or they could be expensive to compute. Moreover, for most regression algorithms the resource usage during the training phase grows linearly or even exponentially with the number of different features used as predictors. Therefore, we add an extra step to our approach, namely a *Feature Selection Algorithm*, which selects only the subset of features which are most relevant to the attack. Feature selection can be performed in many ways. Section 6.1.1 briefly describes how we approached feature selection. After the relevant features are selected, the framework uses this subset of features to build a regression model via a machine learning algorithm.

Note that the proposed approach is not limited to obfuscation and deobfuscation. One can substitute the obfuscation tool in Figure 6.1, with any kind of software protection mechanism (e.g. code layout randomization [167]) and the deobfuscation tool by any known attack implementation corresponding to that software protection mechanism (e.g. ROPeme [135]). This way the set of relevant features and the output prediction model will estimate the strength of the chosen protection mechanism against the chosen attack implementation.

6.1.1. Selecting Relevant Features

Given a set of several software features (e.g. complexity metrics), it is unclear which software features one should aim to change (by applying various obfuscating transformations), such that the resulting obfuscated program is more resilient against certain automated deobfuscation attacks. A conservative approach would be to simply use all available software features in order to build a prediction model. However, this approach does not scale for several regression algorithms (e.g. SVM), because of the large amount of resources needed and also the time needed to train the model. There are several approaches for feature selection published in the literature, e.g. using genetic algorithms [27] or simulated annealing [138]. For our dataset – containing tens of thousands of entries and a few dozen features – we noticed that such feature extraction algorithms are time consuming, i.e. they require weeks of computation time. We have also experimented with principal component analysis (PCA) [170], in order to reduce the number of features. However, this approach did not yield a better prediction accuracy, for our dataset. Therefore, in this section we describe a few light-weight approaches for selecting a subset of features, which are most relevant for a particular deobfuscation attack. The first approach is based on computing correlations and the second approach is based on variable importance in regression models. In Section 6.2 we compare these approaches by building regression models using the features selected by each approach.

First approach: Pearson Correlation

One intuitive way to select relevant features, first proposed by Hall [101], is by computing the Pearson correlation [169] between each of the software features and the attack time. The Pearson correlation is a value in the range $[-1, 1]$. A positive value means that both the time needed for deobfuscation and the software feature tend to have the same increasing

trend, while a negative value indicates that the deobfuscation time decreases as the software feature increases. If the absolute value of this correlation is in: $[0.8, 1]$ then the variables are very strongly correlated, $[0.6, 0.8)$ they are strongly correlated, $[0.4, 0.6)$ moderately correlated, $[0.2, 0.4)$ weakly correlated, $(0, 0.2)$ very weakly correlated and 0 indicates no correlation at all.

After computing the correlation, we sort the features by their absolute correlation values in descending order and store them in a list L. The caveat in selecting the top ten features with the highest correlation is that several of those top ten features may contain couples which are highly correlated with each other. This means that we could discard one of them and still obtain about the same prediction accuracy. To avoid this issue, for each pair of highly correlated features in L, we remove the one with a lower correlation to the deobfuscation attack time. Afterwards, we select the remaining features with the highest correlations.

Second approach: Variable Importance

Another way of selecting relevant features from a large set of features is to first build a regression model (e.g. via random forest, support vector machines, neural networks, etc.), using all available features and record the prediction error. Check the importance of each variable (i.e. feature) using the technique described in [36], i.e. add random noise by permuting values for the i-th variable and average the difference between the prediction error after randomization and before. Repeat this for all $i = \{1, \ldots, n\}$, where n is the total number of variables. Rank the variables according to their average difference in prediction error, i.e. the higher the prediction error, the more important the variable is for the accuracy of the regression model.

Similarly, to the previous approach based on Pearson correlation, we select those features which have the highest importance. In order to reduce over-fitting the regression model to our specific dataset, we employ 10-fold-cross-validation, i.e. the dataset is partitioned into 10 equally sized subsets, training is performed on 9 subsets and testing is performed on the remaining subset, for each combination of 9 subsets. Variable importance is averaged over all of these 10 regression models. Then the features are ranked according to their average importance, i.e. difference in prediction error when the values of that variable are permuted. This procedure is called *recursive feature elimination* [98].

6.2. Case-Study

This section presents a case-study in which we evaluate the approach proposed in Section 6.1. We are interested in answering the following research questions:

RQ1 Which features are most relevant for predicting the time needed to successfully run the symbolic-execution attack presented in Chapter 5?

RQ2 Which regression algorithms generate models that can predict the attack effort with the lowest error?

We focus on the deobfuscation attack based on symbolic execution presented in [21], which is equivalent to extracting a secret license key hidden inside the code of the program via obfuscation. However, in future work we plan to apply the approach proposed in Section 6.1, to other types of automated attacks, such as control-flow simplification [228]. Note that even for other attacks the work-flow from Figure 6.1 remains unchanged, only the details of the attacks and the software features will change.

6.2.1. Experimental Setup

All steps of the experiment were executed on a physical machine with a 64-bit version of Ubuntu 14.04, an Intel Xeon CPU having 3.5GHz frequency and 64 GB of RAM. Subsequently we describe the tools that we have used and how we have used them. The following subsections correspond to the top oval and the rectangles in Figure 6.1 (top to bottom).

Dataset of Original Programs

We have used the same dataset of 4608 programs generated by the `RandomFuns` feature of Tigress, presented in Section 5.2.3. Since this set of 4608 programs might seem too homogeneous for building a regression model, we used another set of 11 non-cryptographic hash functions [168] in our experiments. Similarly to the randomly generated functions, these hash functions, process the input string passed as argument to the program and it compares the result to a fixed value. In the case of the hash functions we print a distinctive message on standard output whenever the input argument is equal to "my_license_key" in ASCII characters. To increase the number of programs in this set, we generated 275 different variants for each of the non-cryptographic hashes using combinations of multiple obfuscation transformations. The point which we aim to show here is that even if we add a small heterogeneous subset to our larger homogeneous set of programs, the smaller subset is going to be predicted with the same accuracy as the programs from the larger set.

Table 6.1a shows the minimum, median, average and maximum values of various code metrics of only the original (un-obfuscated) non-cryptographic hash functions, as computed by the UCC tool and the total number of LOC. Each metric was computed on the entire C file of each program, which includes the hash function and the main function.

Obfuscation Tool

In Chapter 5 we have seen that the similar transformations between Tigress and Obfuscator-LLVM have a very similar behavior w.r.t. symbolic execution. Therefore, in this chapter we have used only the five obfuscating transformations offered by Tigress [56], in order to avoid redundancy. We generate five obfuscated versions of each of the 4608 randomly

Code Metric	Min	Med	Avg	Max
Calculations	4.00	6.00	6.45	12.00
Conditionals	3.00	3.00	3.27	4.00
Logical	2.00	6.00	5.36	11.00
Assignment	8.00	9.00	9.91	16.00
L1.Loops	1.00	1.00	1.00	1.00
L2.Loops	0.00	0.00	0.00	0.00
Total LOC	18.00	25.00	25.99	44.00
Average CC	2.00	2.00	2.14	2.50

(a) Before obfuscation

Code Metric	Min	Med	Avg	Max
Calculations	18.00	27.00	127.70	350.00
Conditionals	3.00	10.00	100.81	444.00
Logical	2.00	6.00	54.45	240.00
Assignment	11.00	17.00	217.36	963.00
L1.Loops	1.00	1.00	1.02	2.00
L2.Loops	0.00	0.00	0.36	3.00
Total LOC	35.00	61.00	501.70	2002.00
Average CC	1.50	3.33	18.80	76.00

(b) After obfuscation

Table 6.1.: Overview of programs containing simple hash functions.

generated programs. The obfuscating transformations we have used are the same as in Chapter 5:

- *Opaque predicates*: introduce branch conditions in the original code, which are either always true or always false for any possible program input. However, their truth value is difficult to learn statically.
- *Literal encoding*: replaces integer/string constants by code that generates their value dynamically.
- *Arithmetic encoding*: replaces integer arithmetic with more complex expressions, equivalent to the original ones.
- *Flattening*: replaces the entire control-flow structure by a flat structure of basic blocks, such that it is unclear which basic block follows which.
- *Virtualization*: replaces the entire code with bytecode that has the same functional semantics and an emulator which is able to interpret the bytecode.

We obfuscated each of the generated programs using these transformations with all the default settings (except for opaque predicates where we set the number of inserted predicates to 16), we obtained $5 \times 4608 = 23040$ obfuscated programs. We obfuscated each of the non-cryptographic hash functions with every possible pair of these 5 obfuscation transformations and obtained $25 \times 11 = 275$ obfuscated programs.

Table 6.1b shows the minimum, median, average and maximum values of various code metrics of the obfuscated set of obfuscated programs involving simple hash functions, as computed by the UCC tool. Each metric was computed on the entire C file of each program, which includes the randomly generated function, the main function and other functions generated by the obfuscating transformation which is applied. For instance, the encode literals transformation generates another function which dynamically computes the values of constants in the code using a switch statement with a branch for each constant. Due to this reason we also notice that after applying the encode literals transformation to a program, its average cyclomatic complexity (CC) is slightly reduced because this function has CC=1 and it is averaged with two other functions with higher CCs.

Deobfuscation Tool

Since all of the original programs print a distinctive message (i.e. "You win!") when a particular input value is entered, we can define the deobfuscation attack goal as: *finding an input value that leads the obfuscated program to output "You win!", without tampering with the program*. As presented in Chapter 5, this deobfuscation goal is equivalent to finding a hidden secret key and can be achieved by employing an automated test case generator. Since we have the C source code for the obfuscated programs, we chose to use KLEE as a test case generator in this study; also KLEE was the fastest symbolic execution engine in the experiments presented in Chapter 5. We ran KLEE with a symbolic argument length of 5 characters, on all of the un-obfuscated and obfuscated programs generated by our code generator, for 10 times each. All of the symbolic executions successfully generated a test case where the input was "12345", which is the input needed to achieve the attacker goal. Similarly we ran KLEE with a symbolic argument length of 16 characters, on all of the un-obfuscated and obfuscated non-cryptographic hash functions, for 10 times each. Again the correct test cases were generated on all symbolic executions, but this time the input was "my_license_key". Note that this is only one way to attack an obfuscated program, and that it does not produce a simplified version of the obfuscated code as in [228]. Rather, it extracts a hidden license key value from the obfuscated code. We computed the mean (M) and the standard deviation (SD) of the reported times across all the 10 runs of KLEE and obtained that 83% of the programs have a relative standard deviation ($RSD = SD/M$) under 0.25 and 94% have $RSD \leq 0.50$. This means that the difference between multiple runs of KLEE on the same program is small.

Software Feature Extraction Tools

In Chapter 5 we saw that the type of operators used by the program have an impact on the time needed to symbolically analyze a program, because these operators affect the path constraints, hence they also affect the query sent to the SMT solver. Other papers [154, 191, 2] also suggest that the complexity of path constraints is a program characteristic with high impact on symbolic execution. However, these papers do not clearly indicate how this complexity should be measured. One way to do this is by first converting the C program into a *boolean satisfiability problem* (SAT instance), and then extracting features from this SAT instance. There are several tools that can convert a C program into a SAT instance, e.g. the C bounded model checker (CBMC) [52] or the low-level bounded model checker (LLBMC) [153], etc. However, the drawback of these tools is that the generated SAT instances may be as large as 1GBs even for programs containing under 1000 lines of code, because they are not optimized. Hence, for our dataset, the generated SAT instances would require somewhere in the order of 10TBs of data and several weeks of computational power, which is expensive.

Instead, we took a faster alternative approach for obtaining an optimized SAT instance from a C program, which we describe next. KLEE generates a *satisfiability modulo theories*

Difficult execution path

Community

Intra-community
Inter-community
Modularity

(SMT) instance for each execution path of the C program. We selected the SMT instance corresponding to the *difficult execution path*, i.e. the path that prints out the distinctive message on standard output. These SMT instances (corresponding to the difficult path), were the most time consuming to solve by KLEE's SMT solver, STP [89]. Many SMT solvers including Microsoft's Z3 [68], often internally convert SMT instances to SAT instances in order to solve them. Hence, we modified the source code of Z3 to output the internal SAT instance, which was saved in separate files for each of the programs in our dataset. For extracting features from these SAT instances we used SATGraf [162], which computes graph metrics for SAT instances, where each node represents a variable and there is an edge between variables if they appear in the same clause (see Figure 6.4). A *community* in a graph is a set of nodes which have a high number of connections between them and a lower number of connections with other nodes of the graph. Connections between the node of a community are called *intra-community* edges, while connections to nodes outside the community structure are called *inter-community* edges. To measure the degree of division of a network into communities, the *modularity* metric (also called Q value) is used. Graphs with a high number of intra-community edges and a low number of inter-community edges have a high modularity. The modularity decreases as the number of intra-community edges decreases and the number of inter-community edges increases. SATGraf computes features such as the number of community structures in the graph, their modularity, and also the minimum, maximum, mean and standard deviation of nodes and edges, inside and between communities. Such features have been shown to be correlated with the difficulty of solving SAT instances [161]. Since symbolic execution includes many queries to an SMT/SAT solver, as shown in [18], these features are expected to be good predictors for the time needed for a symbolic execution based deobfuscation attack. In sum, we transform the path that corresponds to a successful deobfuscation attack into a SAT instance (via an SMT instance), and then compute characteristics of this formula, to be used as features for predicting the effort of deobfuscating the program.

For computing source code features often used in software engineering, on both the original and obfuscated programs, we used the UCC tool [163]. For the programs in our dataset the last four metrics are all zeros, therefore, in our experiments we only used the other eleven metrics, including the total number of LOC. Additionally, we also propose using four other program features, namely: the execution time of the program, the maximum RAM usage of the program, the compiled program file size and the type of obfuscating transformation.

In total we have 64 features out of which 49 SAT features which characterize the complexity of the constraints on symbolic variables and 15 program features which characterize the structure and size of the code. In the following we show that not all of these features are needed for good prediction results.

6.2.2. Feature Selection Results

This section presents the results for the *Feature Selection Algorithms* presented in Section 6.1.1. However, before selecting the most relevant features, we identify how many features (predictor variables) are needed to get good prediction results. For this purpose we performed a 10-fold-cross validation with linear and random forest (RF) regression models using all combinations of 5, 10 and 15 metrics, as well as a models with all metrics. The results in Figure 6.2 show that using 15 variables is enough to obtain a RF model with *root-mean-squared-error* (RMSE) values, which are as good as those from RF models built using all variables. Similar results were obtained for linear models, except that the overall RMSE was higher w.r.t. that of the RF models. Therefore, in the experiments presented in the following sections, we will only select the top best 15 features in both of the two approaches described in Section 6.1.1.

First approach: Pearson Correlation

After employing the algorithm described in Section 6.1.1, we were left with a set of 25 features, with their Pearson correlation coefficients ranging from 0.4523 to -0.0302. The top 15 metrics in this range are shown in Figure 6.3a. The strongest Pearson correlation of the time needed for running the deobfuscation attack is with the average size of clauses in the SAT instance (*mean_clause*), followed by: the average number of times any one variable is used (*meanvar*), the standard deviation of the ratio of inter to intra community edges (*sdedgeratio*), the average number of intra community edges (*meanintra*), the average number of times a clause with the same variable (but different literals) is repeated (*mean_reused*), the average community size (*meancom*), the number of unique edges (*unique_edges*), the number of variables (*vars*), the standard deviation of the number of inter community edges (*sdinter*), the maximum number of distinct communities any one community links to (*max_community*), the number of communities detected with the online community detection algorithm (*ol_coms*), the maximum ratio of inter to intra community edges within any

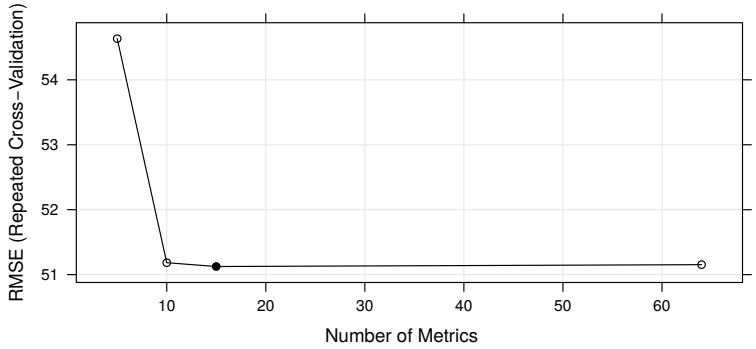
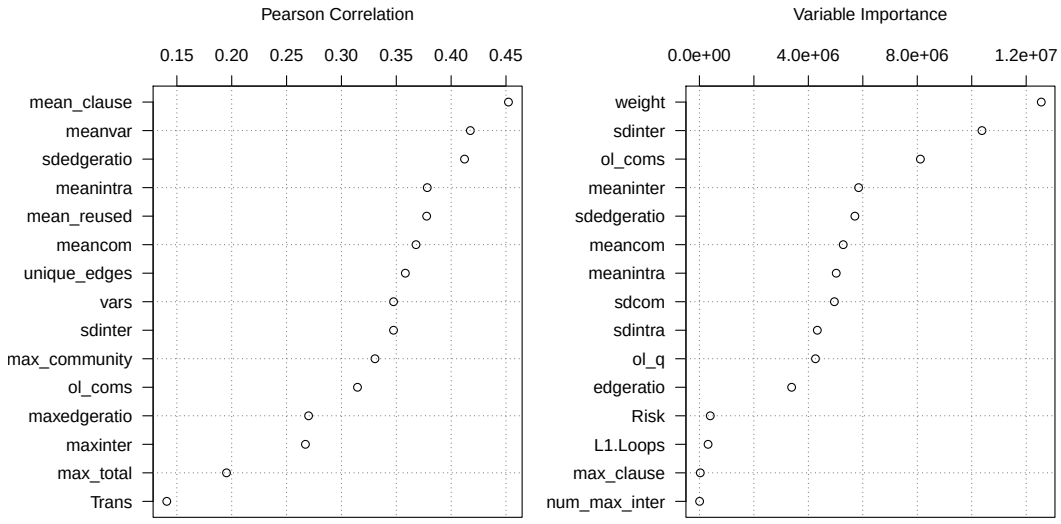


Figure 6.2.: RF models with different feature subsets.



(a) Top 15 features via first approach. (b) Top 15 features via second approach.

Figure 6.3.: Feature Selection Results

community (*maxedgeratio*), the maximum number of inter community edges (*maxinter*), the maximum number of edges in a community (*max_total*) and finally the type of obfuscation transformation employed.

None of the previous features are very strongly correlated to deobfuscation time. The first three features are moderately correlated, the following ten features are weakly correlated and finally the last two features are very weakly correlated. However, notice that the top fourteen features are all SAT features, and none are code metrics from the UCC tool or program features such as execution time, memory usage or file size.

Second approach: Variable Importance

To rank our features according to variable importance we performed recursive feature elimination via random forests, as indicated in Section 6.1.1. Figure 6.3b shows the top 15 features sorted by their variable importance. The features selected using this approach are quite different from those selected in Section 6.2.2. The common features between these two approaches are: *sdinter, ol_coms, sdedgeratio, meancom* and *meanintra*. The first two common features are ranked 2nd and 3rd according to variable importance, however, the most important feature w.r.t. variable importance is the weight of the graph (*weight*), computed as the sum of positive literals minus the sum of negative literals. The 4th most important variable in Figure 6.3b is the average number of inter community edges (*meaninter*), followed by: *sdedgeratio, meancom, meanintra* (see descriptions of these 3 features in Section 6.2.2), the standard deviation of community sizes (*sdcom*), the standard deviation of intra community edges (*sdintra*), the modularity of the SAT graph structure (*ol_q*), the overall ratio of inter to

intra community edges (*edgeratio*), the category of the McCabe cyclomatic complexity [149] (*Risk*), the number of outer-loops (*L1.Loops*), the size of the longest clause (*max_clause*) and the number of communities that have the maximum number of inter community edges (*num_max_inter*).

Similarly, to the first approach, the majority of selected features are SAT features. The only two features which are not SAT features: *Risk* and *L1.Loops* are computed by the UCC tool. The number of loops was indeed indicated also in [18] as being an important feature. The *Risk* has four possible values depending on the value of the cyclomatic complexity (CC), i.e. low if CC $\in [1, 10]$, moderate if CC $\in [11, 20]$, high if CC $\in [21, 50]$ and very high if CC is above 50. CC gives a measure of the complexity of the branching structure in programs (including if-statements, loops and jumps). However, it is remarkable that the CC value was ranked lower than the *Risk*.

Insights from Feature Selection Results

SAT features are important for symbolic execution, because most of the time of the attack is spent waiting for the SAT solver to find solutions for path constraints [18]. Taking a closer look at the common SAT features of both feature selection approaches, we can characterize those SAT instances, which are harder to solve. The graph representation of such an instance has a large number of *balanced* community structures, i.e. a similar number of intra- and inter-community edges. On the other hand, easy to solve instances tend to have *established* community structures, i.e. many more intra-community, than inter-community edges. To check this observation, we downloaded the Mironov-Zhang [154] and the Li-Ye [137] benchmark suites for SAT solvers, containing solvable versions of more realistic hash functions such as MD5 and SHA. All of these instances had *balanced* community structures.

Figure 6.4 illustrates the graph representation[2] of the SAT instance of the MD5-27-4 hash function of the Li-Ye benchmark suite[137] proposed during the 2014 SAT Competition. It is visible – from the number of yellow dots – that this graph has a high number of variables. More importantly it is also visible that one cannot easily distinguish graph community structures, because they are relatively small and well connected with other communities. This kind of structure is hard to solve, because each assignment of a variable has a large number of connections and therefore ramifications inside the graph at the time when *unit propagation* is performed by the SAT solver. However, note that if the graph is fully connected, then it is easy to solve. Therefore, there is a fine line between having too many connections and too few, where the difficulty of SAT instances increases dramatically. This last observation is similar to the *constrainedness of search* employed by Gent et al. [92], when analyzing the likelihood of finding solutions to different instances of the same search problem. This makes sense since a SAT solver is executing a search when it is trying to solve a SAT instance.

[2]These graph representations were generated using the SATGraf tool [162].

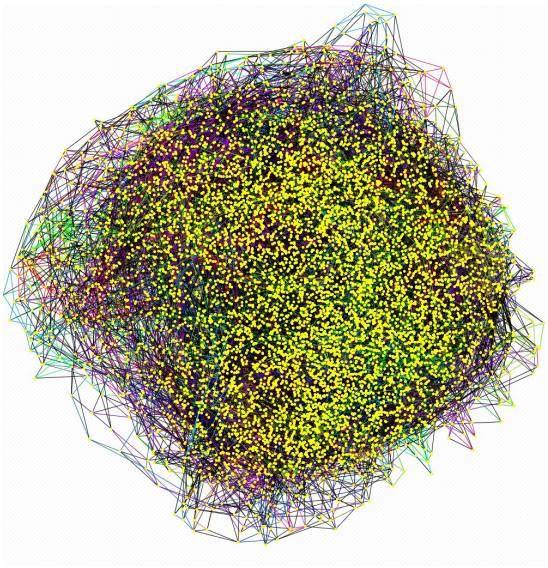


Figure 6.4.: Graph representation of SAT instance corresponding to an MD5 hash with 27 rounds. Solving this instance takes approximately 25 seconds on our testbed.

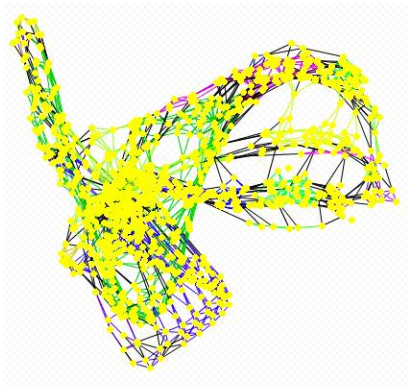
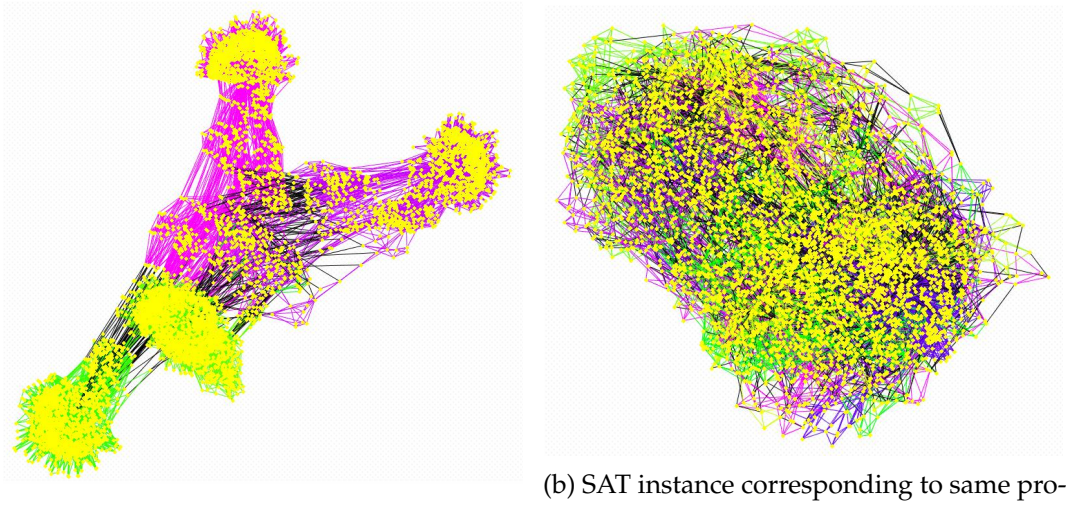


Figure 6.5.: Graph representation of SAT instance corresponding to a program whose symbolic execution time is under 1 second.

On the other hand, many of our randomly generated C programs which were fast to deobfuscate, had *established* community structures. Figure 6.5 illustrates the graph representation of a program generated using our C code generator. This program was generated with the following parameter values:

- `RandomFunsTypes` was set to int, which means that the variable types are integer.
- `RandomFunsForBound` was set to a constant value, which means that the bound does not depend on any symbolic variables, which reduces the number of branches of



(a) SAT instance corresponding to a program containing a non-cryptographic hash function, before obfuscation. This instance is solved in about 7.5 seconds.

(b) SAT instance corresponding to same program from Figure 6.6a, after being obfuscated with *virtualization* and *flattening*. This instance is solved in about 438 seconds.

Figure 6.6.: The Effect of Obfuscation on SAT Instances.

the search tree.

- `RandomFunsOperators` was set to `Shiftlt`, `Shiftrt`, `Lt`, `Gt`, `Le`, `Ge`, `Eq`, `Ne`, `BAnd`, `BOr` and `BXor`, which is the set of logical and bitwise operators, that were shown to be fastest to solve in Chapter 5.

- `RandomFunsControlStructures` was set to (if (if (if (bb n) (bb n)) (if (bb n) (bb n))) (if (bb n) (bb n))), which consists only of if statements, hence the search tree is not deep.

- $n = 1$, which means that all basic blocks consist of only one statement.

- `RandomFunsPointTest` was set to true, which means that a license check is added to the program in the form of an if statement.

This instance is expected to be fast to solve, because it does not involve any loops dependent on symbolic inputs and it only involves logical and bitwise operators. Balanced community structures translate to a high diffusion of the symbolic input to output bits, i.e. affecting any bit of the input license key will affect the result of the output, which is the case for collision-resistant hash functions, as well as the effect of obfuscation transformations like *virtualization*, *flattening* and *arithmetic encoding*.

	SVM	RF	GP	NN
UCC (11 features)	0.019	0.016	0.018	0.018
Pearson (15 features)	0.017	0.013	0.015	0.015
Var. Importance (15 features)	0.019	0.013	0.015	0.015

Table 6.2.: The NRMSE between model prediction and ground truth (average over NRMSE of 10 models)

The Effect of Obfuscation on SAT Instances

In this context of representing SAT instances as graphs, it is interesting to note the effect of obfuscation transformations on SAT instances. For instance, Figure 6.6a illustrates the SAT instance of a non-obfuscated, non-cryptographic hash function from our dataset. The community structures of this hash function are *established*, hence, the instance can be solved in about 7.5 seconds. However, after applying two layers of obfuscation, first using the *virtualization* and then the *flattening*, transforms the SAT instance of this program into the one illustrated in Figure 6.6b. This instance, has a *balanced* community structure, hence, slower to solve (438 seconds) and shares a resemblance to the MD5 instance from Figure 6.4. We have also noticed that the *arithmetic encoding* transformation has this effect on SAT instance. However, the *opaque predicate* and *literal encoding* alone do not have such an effect.

6.2.3. Regression Results

For each of the regression algorithms presented next, we have used several different configuration parameters. Due to space limitations, we only present the configuration parameters which gave the best results. We randomly shuffled the programs in our 2 datasets of programs into one single dataset and performed 10-fold cross-validation for each experiment. To interpret the *root-mean-squared-error* (RMSE) we normalize it by the range between the fastest and slowest times needed to run the deobfuscation attack on any program from our dataset. Since our dataset contains outliers (i.e. either very high and very low deobfuscation times), the normalized RMSE (NRMSE) values are very low for all algorithms, regardless of the selected feature subsets, as shown in Table 6.2. This could be misinterpreted as extremely good prediction accuracy regardless of the regression algorithm and feature set. However, we provide a clearer picture of the accuracy of each regression model by computing the NRMSEs after removing 2% and 5% of outliers from both the highest and the lowest deobfuscation times in the dataset. This means that in total we remove 4%, respectively 10% of outliers. Instead of showing just the numeric values of the NRMSE for each these three cases (0%, 4% and 10% of outliers removed), we show cumulative distribution functions of the relative (normalized) error in the form of line plots, e.g. Figure 6.7. These line plots show the maximum and the median errors for all the three cases, where the x-axis represents the percentage of programs for which the relative error (indicated on the y-axis) is lower than the plotted value.

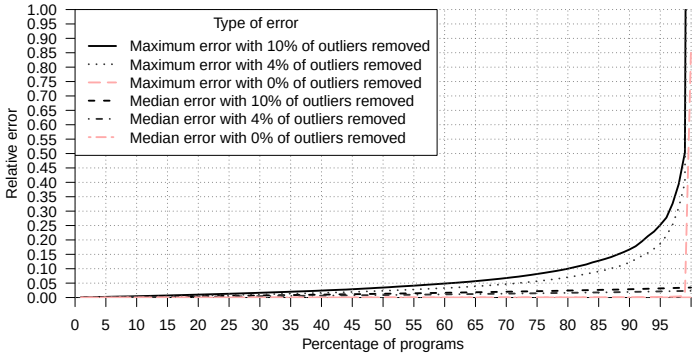


Figure 6.7.: Relative prediction error of RF model.

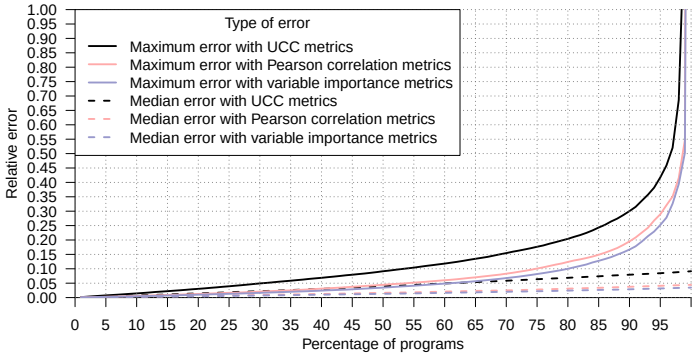


Figure 6.8.: RF models with different feature sets.

Note that in addition to the following regression algorithms we have also employed both linear models and generalized linear models [160]. However, the results of the models generated by these algorithms were either much worse compared to the models presented in the following, or the models did not converge after 24 hours.

Random Forests (RFs)

Random forests (RFs) were proposed by Breiman [36] as an extension of random feature extraction, by including the idea of "bagging", i.e. computing a mean of the prediction of all random decision trees. In our experiments we constructed a RF containing 500 decision trees.

Figure 6.7 shows the maximum and median relative errors for 0%, 4% and 10% of outliers removed. As more outliers are removed the relative error increases due to a decrease in the range of deobfuscation times in the dataset. However, even when 10% of outliers are removed, the maximum error is under 17% and the median error is less than 4% for 90% of the programs, which seems acceptable for most use cases.

Note that the model in Figure 6.7 was built using the 15 features selected via variable

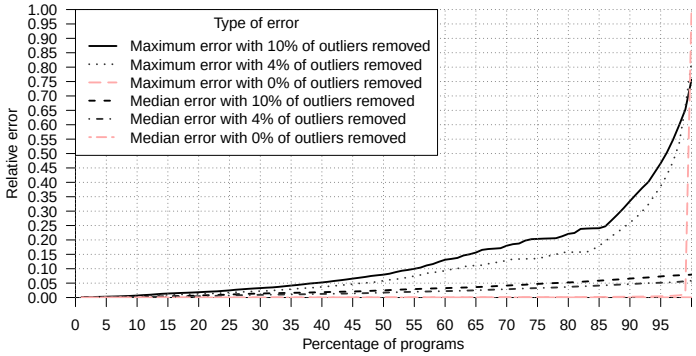


Figure 6.9.: Relative prediction error of SVM model.

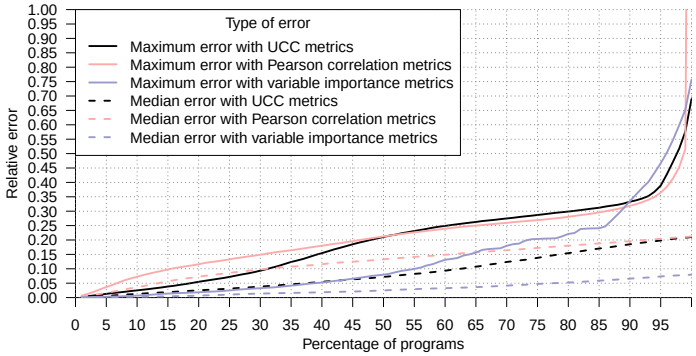


Figure 6.10.: SVM models with different feature sets.

importance, presented in Section 6.2.2. We chose to show the results from the model built using these features because, they are better than those produced by models built using other subsets of features. As we can see from Figure 6.8, the relative error values when building models with UCC metrics only and with the Pearson correlation approach, give worse results in terms of both maximum and median error rates.

Support Vector Machines (SVMs)

Support vector machines (SVMs) were proposed by Cortes and Vapnik [61] to classify datasets having a high number of dimensions, which are not linearly separable.

Figure 6.9 shows the relative errors for the SVM model built using the features selected via the second approach (see Section 6.2.2). The accuracy of this model is lower than the RF model from Figure 6.7, i.e. the maximum relative error is just below 35% for 90% of the programs, when we remove 10% of the outliers. However, the median error is less than 7% in the same circumstances. The reason why SVM performs worse than RF is due to the bagging technique applied by RF, whereas SVM uses a single non-linear function.

Again we chose to show the SVM model built using the features selected via variable

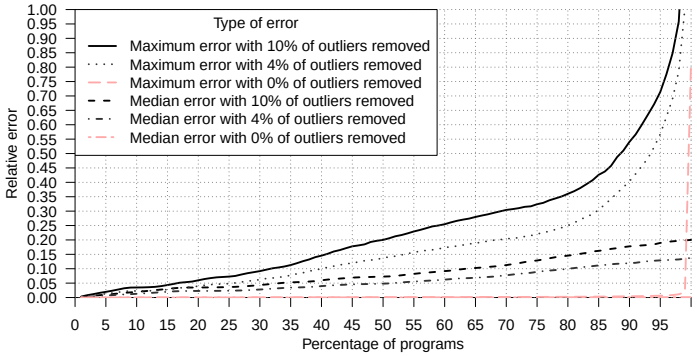


Figure 6.11.: Relative prediction error of GP model.

importance in Figure 6.9, because, as we can see from Figure 6.10, the maximum and median error rates for this model are much lower than the SVM models built using only UCC metrics or the features selected via Pearson correlation. Note that the maximum error of the model built using variable importance surpasses that of the other two models around the 90% mark on the horizontal axis. This means that for 10% of the programs the maximum error of the model built using the features selected by variable importance, is higher that the error of the other two models. However, note that the median error is around 10% lower in the same circumstances.

Genetic Programming (GP)

Given the set of all code features as a set of *input variables*, GP [108] searches for models that combine the input variables using a given *set of functions* used to process and combine these variables, i.e. addition, multiplication, subtraction, logarithm, sinus and tangent in our experiments. GP aims to optimize the models such that a given *fitness function* is minimized. For our experiments, we used the *root-mean-square error* (RMSE) between the actual time needed for deobfuscation and the time predicted by the model, as a fitness function. The output of GP is one of the generated models with the best fitness value. In our case this member is a function of the code features, which has the smallest error in predicting the time needed to execute the deobfuscation attack on every program. For instance, the best GP model built using the features selected via variable importance is shown in equation 6.1:

$$\begin{aligned} time = {} & (\text{edgeratio} + cos(\text{ol\_coms}) + cos(cos(\text{sdcom} + \text{num\_max\_inter}) + \text{L1.Loops})) \\ & * (\text{sdinter} * (\text{sdedgeratio} - sin(\text{meanintra} * -1.27))) \\ & * (\text{sdedgeratio} - sin(\text{meanintra} * -1.27)) * (1.03 - sin(0.04 * \text{sdinter})) * \text{sdedgeratio} + 10.2. \end{aligned} \tag{6.1}$$

Note that only seven distinct features were selected by the GP algorithm for this model, from the subset of 15 features. Figure 6.11 shows the maximum and median error values for the GP model from equation 6.1. Note that the maximum and median error levels for the dataset where 10% of outliers are removed, are 55%, respectively 19% for 90% of the

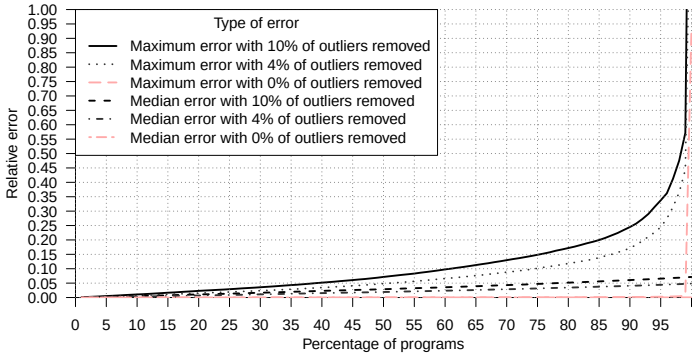


Figure 6.12.: Relative prediction error of NN model.

	SVM	RF	GP	NN
UCC (11 features)	370	499	12.5	0.48
Pearson (15 features)	2094	651	7.9	0.50
Var. Importance (15 features)	2094	651	7.3	0.50

Table 6.3.: Size of the prediction models (in MBs).

programs. This error rate is much higher than both RFs and SVMs and is due to the fact that the GP model is a single equation.

Neural Networks (NNs)

Multi-layer neural networks (NNs) were introduced by Werbos [221] in the 1970s. Recently, the interest in NNs has been revived due to the increase in computational resources available in the cloud and in graphical processing units. A neural network has three characteristics. Firstly, the *architecture* which describes the number of neuron layers, the size of each layer and connection between the neuron layers. In our experiments we used a NN with five hidden layers each containing 200 neurons. The input layer consists of the set of code features and the output of the NN is a single value that predicts the time needed to run the deobfuscation attack on a program. Secondly, the *activation function* which is applied to the weighted inputs of each neuron. This function can be as simple as a binary function, however it can also be continuous such as a Sigmoid function or a hyperbolic tangent. In our experiments we use a ramp function. Thirdly, the *learning rule* which indicates how the weights of a neuron's input connections are updated. In our experiment we used the *Nesterov Accelerated Gradient* as a learning rule.

Figure 6.12 shows the maximum and median error of the NN model built using all metrics. Note that in the case of NNs it is feasible to use all metrics without incurring large memory usage penalties such as is the case for SVMs. The performance of this model is better than the SVM and GP models, but not better than the RF model.

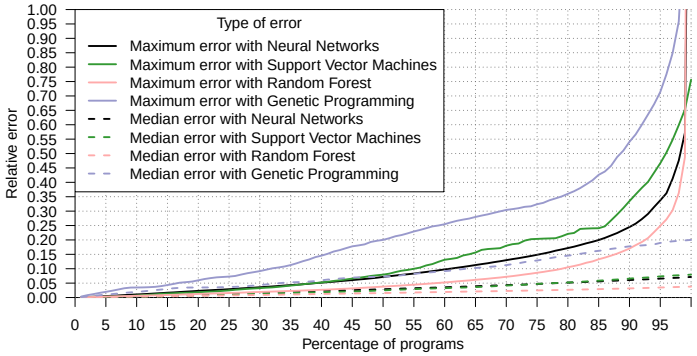


Figure 6.13.: Comparison of regression algorithms.

6.3. Summary and Threats to Validity

Based on the results presented above, we answer the research questions elicited in the beginning of Section 6.2. Firstly, in Figure 6.2 we have seen that given our large set of 64 program features, using only 15 is enough to obtain regression models with RMSEs as low as the regression models where all the features are used. From Figures 6.3a and 6.3b we have seen that both approaches to feature selection ranked SAT features above code metrics commonly used to measure resilience, namely cyclomatic complexity or the size of the program. This means that the most important characteristics for symbolic execution based attacks is the complexity of the path constraints involving symbolic variables. The reason why SAT features have a higher impact on symbolic execution is that most of the time during symbolic execution is spent waiting for the SMT solver and these features indicate the time that is needed by the SMT solver to find a counter example for path constraints.

Secondly, Table 6.2 shows the RMSE for different regression models normalized by the fastest and slowest deobfuscation attacks in our dataset. Since our dataset contains outliers, the results from Table 6.2 are misleading. Therefore we removed 4% and 10% of the outliers from our dataset and plotted the cumulative distribution of the errors for each of the regression models. From Figures 6.8 and 6.10 we observe that the second approach to feature selection, based on variable importance, gives better results than the first approach, based on Pearson correlation. Therefore, in Figure 6.13 we plot the maximum and median errors of the models from the four different regression algorithms, where 10% of outliers are removed from the dataset. From the first glance at Figure 6.13, one may conclude that RF has the lowest overall maximum error rate, followed by NN, SVM and GP. However, the median error of the RF, NN and SVM models are all lower than 8% for all programs. This indicates that if the median error is the key performance indicator, it is much less important whether we pick RF, NN or SVM as the regression algorithm. Another observation is that the size of the prediction models from RF models are generally smaller than those of SVM models as seen in Table 6.3. However, models obtained from GP and NN are one, respectively two orders of magnitude smaller than RF models. The size of SVM, RF and GP

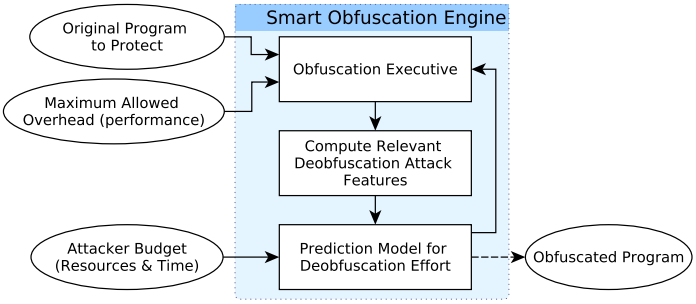


Figure 6.14.: Combining results with obfuscation tools.

models grows proportionally to the number of features used. An advantage of NN models is their relatively small size of around 50 Kilobytes is constant for any number of features used. This is understandable because the number of weights and neurons is negligibly influenced by the number of features used to build the model.

In sum, the most relevant features for characterizing the deobfuscation attack based on symbolic execution, are SAT features (RQ1). Moreover, the regression algorithm which yields the highest prediction accuracy is random forest (RQ2).

These results can be used to build the *Smart Obfuscation Engine* (SObE) shown in Figure 6.14, where the ovals represent inputs and outputs. SObE takes three inputs: (1) the original program source code, (2) the maximum allowed performance overhead of the resulting obfuscated program and (3) the resource and time available to the attacker (attacker budget). SObE first gives the original program to the *Obfuscation Executive* (OE) [104]. The OE proposed by Heffner and Collberg [104], uses software complexity metrics and performance measurements to choose a sequence of obfuscating transformations, that should be applied to a program in order to increase its potency. Therefore, the OE in Figure 6.14 applies a set of obfuscation transformations that satisfy the maximum allowed overhead. Afterwards, SObE computes the relevant features (determined in Section 6.2.2) on the obfuscated program and then uses the best prediction model from Section 6.2.3 to estimate the effort needed by the deobfuscation attack. If the effort is less than the attacker's budget, then this is signaled to the OE and the process restarts, otherwise the obfuscated program is output.

Threats to Validity In our case study, we have generated a dataset of unobfuscated programs of up to a few 100s of LOC. Obfuscating these programs generates programs having up to a few 1000s of LOC. Therefore, the regression models generated in the case study may not be accurate for all possible programs. In our experiments we have found that the size of the program is very weakly correlated with the time needed to run the deobfuscation attack based on symbolic execution. This is slightly counter-intuitive because, one may think that more LOC implies more complex path constraints due to additional computations. In reality, the length of the execution trace is more important than the LOC,

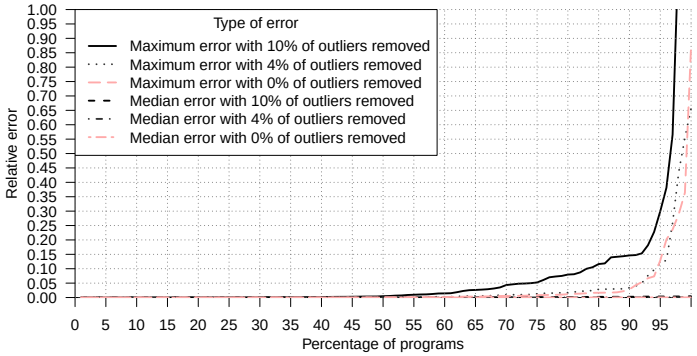


Figure 6.15.: Relative error of hash functions only.

i.e. even programs with less than 100 LOC may have loops which execute for millions of iterations, as opposed to programs with 1000s of LOC, which do not contain loops with such a high number of iterations.

The experiments in this chapter mainly focus on relatively small programs, because of three issues of symbolic execution (which will be discussed in more detail in Section 7.2), i.e.: (1) an explosion of the number of paths to be analyzed in the program, (2) a divergence between the operational semantics of the program and the semantics implemented by the symbolic execution engine and (3) path constraints which require a long processing time by an SMT-/SAT-solver. These issues often arise when analyzing real-world programs, which often employ OS API calls and third party library calls. Such functions generally contain intricate control-flow structures to handle all possible inputs and errors. Moreover, these functions highly likely make calls to other OS API functions or third party library functions, a.s.o. Therefore, a program containing as few as 10 LOC, may cause a path explosion if this program calls functions to: (1) read the contents of a file, whose name is indicated by a symbolic variable, (2) archives the contents and (3) sends the result to an IP address indicated by another symbolic variable. In this thesis we have limited the number of function calls in the programs from our datasets (i.e. the only third library functions called were printf, atoi and strtol), to be able to control the type and amount of code that will be symbolically executed. Therefore, when applying our prediction model to real-world programs it is necessary to inline any third party library functions that we have not used. Moreover, our prediction model was built for programs, which were successfully attacked by the KLEE symbolic execution engine before the cutoff time of 1 hour was reached. This means that our prediction model may underestimate the resilience of more complex (obfuscated) programs against symbolic execution. However, the value predicted by our model – in the case of such complex programs – will be a conservative estimation of the resilience against symbolic execution attacks.

We tested the prediction accuracy of our best RF model (from Figure 6.7) when including a small non-artificial dataset of programs containing non-cryptographic hash functions. Figure 6.15 indicates that the prediction error of our best RF model (trained using 10-fold-

Instance Name	Solver(s)	Predicted(s)	$\frac{Predicted}{Solver}$
MD5-27-4	25.37	71.56	2.82
mizh-md5-47-3	681.29	950.43	1.39
mizh-md5-47-4	235.53	1069.19	4.53
mizh-md5-47-5	1832.96	437.98	0.23
mizh-md5-48-2	445.19	523.70	1.17
mizh-md5-48-5	227.05	644.38	2.83
mizh-sha0-35-2	330.48	158.57	0.47
mizh-sha0-35-3	139.93	213.03	1.52
mizh-sha0-35-4	97.62	214.61	2.19
mizh-sha0-35-5	164.71	193.49	1.17
mizh-sha0-36-2	85.44	222.07	2.59

Table 6.4.: Prediction results of realistic hash functions via RF model trained with SAT features from Section 6.2.2. The solver and predicted time are given in seconds.

cross-validation on both datasets), for the samples in the smaller dataset alone, has similar levels to the prediction error of the entire dataset.

We also performed a reality check, i.e. we verified that the SAT features we identified are also relevant for the realistic hash functions. For this purpose we have first trained a RF model using only the top 10 most important SAT features from Section 6.2.2, computed on the SAT instances of our dataset or randomly generated programs and non-cryptographic hash function. Afterwards, we have applied this RF model to the Mironov-Zhang [154] and the Li-Ye [137] benchmark suites for SAT solvers, containing solvable versions of more realistic hash functions such as MD5 and SHA. Table 6.4 shows the results obtained from applying the RF model to the hash functions, which were solvable by the *minisat* solver used by STP (KLEE's SMT solver), on our machine. Note that the Li-Ye [137], suite contains many other instances of MD5 with more rounds, however, those could not be solved within a 10 hour time limit on our test machine. The last column of Table 6.4 gives the ratio between the predicted and the actual time needed to solve each instance. Except for the *mizh-md5-47-4* and *mizh-md5-47-5* SAT instances, which are the most over- and respectively under-estimated, the rest of the predictions are quite encouraging, given that we have not trained the RF model with any SAT instances corresponding to MD5 or SHA hash functions.

Other features which we have not used in our work and are yet to be discovered, may further improve prediction. We plan to explore more features in future work. In this work we have focused on a deobfuscation attack based on symbolic execution. Other attacks which will be the topic of future work, may be predicted more accurately. Our choice of feature extraction and regression algorithms is limited, however, we believe we have covered a representative set of algorithms.

7. Improving Obfuscation Transformations Against Symbolic Execution

This chapter presents novel obfuscation transformations, which aim to raise the bar against symbolic execution attacks. These transformations specifically target the program characteristics derived using the search model from Chapter 4. Parts of this chapter have been published in two peer-reviewed publications [18, 20] co-authored by the author of this thesis.

In Chapter 4 we saw that a software developer who wants to protect an application against automated MATE attacks should first identify all likely attacks and describe them using an *attack-net* (see Figure 4.1). Each path from the source(s) to one of the sinks of an attack net, represents one automated MATE attack, which consists of several steps (i.e. transitions of the petri-net). In Section 4.2, we proposed viewing each of the non-trivially solvable transitions, as search problems. Therefore, we argue that the cost of an automated MATE attack is equivalent to the sum of costs of solving each of the search problems (i.e. executing each transition) in that attack.

Since MATE attackers are generally driven by economic incentives, they will most likely choose the automated attack with the lowest cost. In the survey from Section 4.3, we saw that different attack steps (i.e. search problems) have different costs on unprotected (i.e. un-obfuscated) programs. We also mentioned some obfuscation techniques to hamper various attacks. In this chapter we present two novel obfuscation techniques for raising the bar against symbolic execution attacks. These obfuscation techniques were derived based on an observation made in the case-studies about symbolic-execution attacks, presented in Chapter 5. This observation states that existing obfuscation transformations do not add execution paths dependent on symbolic variables, they only add paths dependent on other variables. Therefore, the two transformations we propose both add such paths in different ways.

7.1. The Impact of Obfuscation on Search Problems

Before jumping into the details of the proposed obfuscation techniques, we reiterate over the automated MATE attack-net discussed in Section 4.2, to motivate the contributions of this chapter. Consider the *attack-defense* tree from Figure 7.1, corresponding to the *attack-net* Figure 4.1, representing the set of automated MATE attacks for bypassing license checks

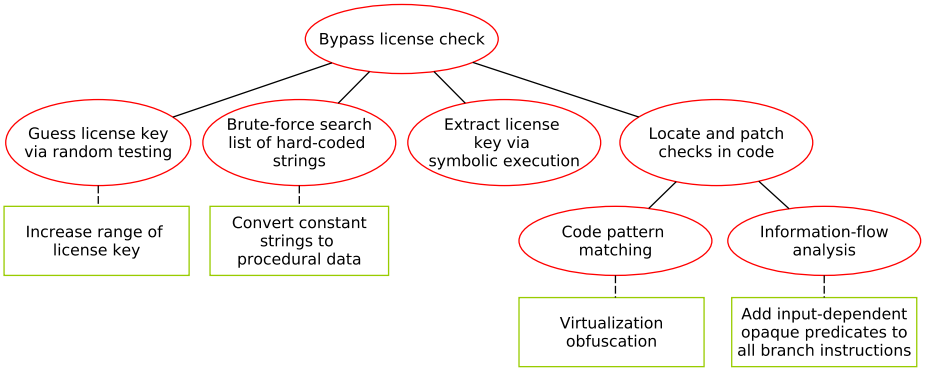


Figure 7.1.: Attack-defense tree corresponding to attack-net from Figure 4.1.

in an application, e.g. a computer game. The ovals in the attack-defense tree represent attacks, while the rectangles represent defenses (i.e. countermeasures for the associated attack node). By successfully executing one of the automated MATE attacks corresponding to the oval-leaves of the tree, the attacker achieves the goal of bypassing the license check.

The license check can be bypassed by employing different types of static and/or dynamic analyses. One way is to dynamically execute the program with all possible inputs, until the password is found (left-most child of the root in the tree from Figure 7.1). However, if the defender increases the range of the license key by using several (e.g. >16) alphanumeric characters and also includes special characters, this guessing attack is considered too costly. The reason is that the search problem corresponding to this attack can be modeled as shown in Table 4.9, which means that the attack tree has a branching factor equal to the number b of different alphanumeric characters and a depth equal to the length of the license key l. Therefore, the total number of leaves in the search tree corresponding to this attack is b^l, which is extremely large if $b > 36$ – because there are 36 alphanumeric characters – and $l > 16$.

Another approach is to extract all hard-coded standard C-strings inside a program and put them inside of a list (second child of the root in the tree from Figure 7.1). Then try each entry from the list of hard-coded strings, as an input to the dynamic program execution, until the key is found. Since this list is likely much smaller than the input space of the program, the search cost is much smaller. A preceding step of this brute-force search for the key in the list of hard-coded strings, is to extract all hard-coded strings from a binary executable. The heuristic for that step is that by definition strings are consecutive sequences of – more than 5 – printable ASCII characters ending with a null byte. This heuristic can be broken by the *convert static data to procedural data* obfuscation transformation presented in Section 3.3. If there defender uses this obfuscation transformation to break strings into single (encoded) characters, which are dynamically (decoded and) stitched together in the correct order, this attack is no longer feasible. Therefore, after applying this obfuscation transformation, the attacker has to either find a better approach or roll-back to a brute-force

on the input space.

A further approach is to locate the checks inside the code and then patch them, such that entering a correct license key is no longer necessary (right-most child of the root in the tree from Figure 7.1). Locating the checks inside the code can be done in two different ways. The first way is to use a static pattern for license checks and identify all instructions in the code that match this pattern. However, such patterns can be erased by employing code obfuscation transformations such as virtualization. The second way to locate checks it by performing information-flow analysis, e.g. via taint analysis on the value given as input to the license key argument. This way, any branch in the code that depends on a tainted value can be considered a license check. Nevertheless, this attack can be hampered by adding opaque predicates based on symbolic values, to every branch in the existing code. These opaque predicates can be easily added such that they do not affect the original functionality of the program. This would cause taint analysis to identify all branches as license checks, which is incorrect.

Therefore, according to the attack-defense tree from Figure 7.1, another way to bypass the license check is to extract the license key value from the code via symbolic execution. One may object that symbolic execution could also be hampered by using a cryptographic hash function on the input license key and comparing the result of the hash with a hard-coded value. However, without applying any code obfuscation transformation on top of this hash function, it would be easy to locate and patch using code pattern matching.

7.2. Existing Anti-Symbolic Execution Obfuscations

Anand et al. [2] indicate 3 fundamental issues of symbolic execution: (1) path explosion, (2) path divergence and (3) complex constraints. Therefore, works that propose obfuscation techniques against symbolic execution focus on exploiting at least one of these fundamental issues, which we discuss in the following.

7.2.1. Path Explosion

Wang et al. [219] propose an approach based on an unsolved mathematical problem, which involves only linear operations on integers, called the Collatz conjecture [91]. This obfuscation implies adding a loop bounded by a symbolic value to an existing program. Such a loop would generate a path explosion for the symbolic execution engine, however, executing it dynamically, it will always converge to a fixed known value, i.e. 1 in the case of the Collatz conjecture. Our work also proposes an approach that causes path explosion, however, in contrast to Wang et al. [219], our approach is not based on unsolved mathematical conjectures.

7.2.2. Path Divergence

Yadegari and Debray [227], Sharif et al. [191] and Cavallaro et al. [42] show that converting explicit control-flow into implicit control-flow hampers white-box test case generators based on taint-analysis and symbolic execution. Path divergence refers to situations where the symbolic execution engine cannot compute precise path constraints from the program code. This leads to a divergence between generated tests and the actual program paths.

The key to transforming explicit to implicit control flow is using symbolic variables for computing an address where the program jumps to unconditionally. This transformation removes any comparisons between the symbolic variables and constants used by symbolic execution engines. In the case of such obfuscation transformations one may employ heuristics based on knowledge about the architecture of the CPU [227].

Another type of obfuscation transformation which is known to cause path divergence is dynamically modifying code [10]. Such code cannot be analyzed statically, because its static image changes at runtime while the code is executing. If the value of the instructions that are being dynamically generated depend on input values, then the symbolic execution engine must guess the right value of the instruction that is to be executed, which is difficult. The drawback of dynamically modifying code is that it opens the door to remote attacks. Since most software developers are more concerned about remote attacks than automatic analysis attacks, they would avoid using such transformations and resort to obfuscation transformations which do not induce this risk. Therefore, in this work we proposed obfuscation transformations which do not introduce remote attacks.

7.2.3. Complex Constraints

Applying cryptographic hash functions to equality checks based on input values is a type of data obfuscation transformation, which is problematic for symbolic execution. In particular, the SMT solvers used by symbolic execution engines are known to have practical limitations w.r.t. inverting cryptographic hash functions [154, 191]. However, we note that cryptographic hash functions are based on large look-up tables containing random numbers which are publicly known and easy to locate in code, if they are not obfuscated using one of the transformations presented in Chapter 3. Therefore, we argue that an active MATE attack will be able to locate and disable explicit checks which only use hash functions on input values. Implicit checks cannot be disabled, however they may cause the application to crash which is undesirable for many developers or even worse they may open the door for remote attacks as discussed in the previous paragraph. Moreover, hash functions are only applicable for equality comparisons and therefore range comparisons need to be protected in a different way. In this chapter we discuss obfuscation transformations that hide the location of checks via control-flow obfuscation.

Listing 7.1: *Range divider* with 2 branches

```
1 if (x > 42)
2   z = x + y + w
3 else
4   z = (((x ^ y) + ((x & y) << 1)) | w) +
5       (((x ^ y) + ((x & y) << 1)) & w);
```

Listing 7.2: Program with loop

```
1  int main(int argc, char* argv[]) {
2    unsigned char *str = argv[1];
3    unsigned int hash = 0;
4    for(int i = 0; i < strlen(str); str++, i++) {
5      hash = (hash << 7) ^ (*str);
6    }
7
8    if (hash == 809267)
9      printf("You win!");
10
11   return 0;
12 }
```

7.3. Proposed Obfuscation Transformations

The proposed obfuscation transformations are inspired by observation 10 in step 3 of the experiment described in Section 5.2.2, i.e. branch instructions added by control-flow obfuscation transformations do not depend on program inputs. Therefore, we propose making branch instructions added by control-flow obfuscation transformations dependent on program inputs to increase the slowdown of symbolic execution by creating a path explosion. Making such branch instructions input dependent may or may not be effective for existing obfuscation transformations. For instance, making the opaque predicates p added by Tigress input dependent, causes KLEE to issue a query that includes the branch condition corresponding to p. However, the SMT solver always determines that p or $\neg p$ is unsatisfiable or it times out before it can find any satisfiable assignment and does not analyze the code in such unfeasible paths. In the following we propose two obfuscation transformations which introduce feasible paths in the original program.

7.3.1. Range Dividers

Range dividers

Our first proposal is an obfuscation transformation called *range divider*. *Range dividers* are branch conditions that can be inserted at an arbitrary position inside a basic block, such that they divide the input range into multiple sets. In contrast to opaque predicates, *range divider* predicates may have multiple branches, any of which could be true and false depending on program input. This will cause a symbolic execution engine to explore all branches of a range divider. In order to preserve the functionality property of an obfuscator, we use equivalent instruction sequences in all branches of a *range divider* predicate, as

Listing 7.3: Program from Listing 7.2 obfuscated with *range divider*

```
1  int main(int argc, char* argv[]) {
2    unsigned char *str = argv[1];
3    unsigned int hash = 0;
4
5    for(int i = 0; i < strlen(str); str++, i++) {
6      char chr = *str;
7      if (chr > 42) {
8        hash = (hash << 7) ^ chr;
9      } else {
10       hash = (hash * 128) ^ chr;
11     }
12   }
13
14   if (hash == 809267)
15     printf("You win!");
16
17   return 0;
18 }
```

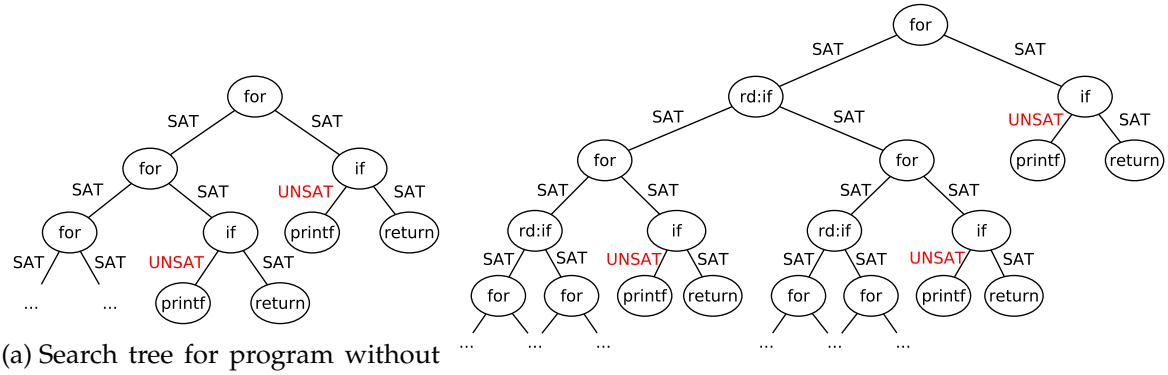
illustrated in Listing 7.1. To prevent compiler optimizations from removing *range divider* predicates, due to equivalent code in their branches, we employ software diversity (on the code of every branch) via different obfuscation configurations, e.g. in Listing 7.1 we used the *EncodeArithmetic* transformation of Tigress, on the *else* branch. We have experimented with all optimization levels of LLVM *clang* and none of them remove *range divider* predicates if their branches are obfuscated. On the downside, *range dividers* increase the size of the program proportionally to the total number of branches.

The effectiveness of a *range divider* predicate against symbolic execution depends on:

1. The number of branches of the predicate, denoted ρ; note that for *switch*-statements $\rho > 2$ and is equal to the number of cases plus the default branch.

2. The number of times the predicate is executed, denoted τ.

More specifically, its number of paths increases according to the function: ρ^{τ}. For example, consider the program from Listing 7.2, which computes a value (`hash`) based on its first argument (`argv[1]`) and outputs "You win!" on the standard output if this value is equal to 809267. It has an execution tree with $2 \times$ `strlen(argv[1])` paths, because it does not contain a *range divider*. The search tree corresponding to applying symbolic execution to this program is illustrated in Figure 7.2a. Note that – for the sake of brevity – the nodes do not contain the entire source code, but just an indication of where the instruction pointer is pointing to. We also only show branch conditions in all internal nodes, the leaves are just the `printf` and the `return` statements. Note that the goal of the search is to find a node where the instruction pointer is at `printf` and the node is *SAT*. The depth of this tree is proportional to the length of the input. The program in Listing 7.3 is obtained by obfuscating the program from Listing 7.2 using *divide range* predicate with $\rho = 2$ branches. The resulting program has $2^{\texttt{strlen(argv[1])}}$ paths, because the predicate



(a) Search tree for program without range divider (Listing 7.2).

(b) Search tree for program with range divider (Listing 7.3).

Figure 7.2.: The effect of range dividers on the search tree.

is executed τ = `strlen(argv[1])` times. The search tree corresponding to applying symbolic execution to this program is illustrated in Figure 7.2b. Note the new nodes labeled *rd:if*, which represents the *if*-statement corresponding to the *range divider* on line 7 of Listing 7.3. Similarly to the search tree from Figure 7.2a, the depth of this tree is proportional to the length of the input, but the number of paths is much larger.

Limitations of range dividers We can further increase the number of paths by adding more branches to the *divide range* predicate from Listing 7.3. However, the number of possible branches is upper-bounded by the cardinality of the type of the variable used in the range divider. In the example from Listing 7.3 the maximum number of branches is 256 because variable `chr` is of type `char`. Therefore, the maximum number of branches in this example is achieved by a *switch*-statement with 256 cases as shown in Listing 7.4. This effectively results in a branching factor of $\rho = 256$ for each iteration of the loop. Hence, the number of paths is $256^{\texttt{strlen(argv[1])}}$. The symbolic execution slowdown induced by *range dividers* is caused by: (1) the increase in the number of paths, but also (2) the type of obfuscation transformations applied on each branch of each *range divider*.

7.3.2. Input Invariants

The *divide range* obfuscation transformation proposed in Section 7.3.1 may induce a high increase in the effort needed by symbolic execution engines. However, this depends on the code which is being obfuscated. If the code does not include any loops (e.g. the program from Listing 7.5), then obfuscating with *range dividers* will not induce a significant slowdown of symbolic execution attacks. Therefore, in this section we propose obfuscation transformations which are able to obfuscate even the simplest code, however, with the cost of changing the exact input-output semantics of the program. That is, we deliberately violate the *functionality* property of the obfuscator definition of Barak et al. [26], which states

Listing 7.4: Program from Listing 7.2 obfuscated with maximum number of branches of *range divider*

```
int main(int argc, char* argv[]) {
  unsigned char *str = argv[1];
  unsigned int hash = 0;

  for(int i = 0; i < strlen(str); str++, i++) {
    char chr = *str;
    switch (chr) {
      case 1: hash = (hash << 7) ^ chr;
        break;
      case 2: // obfuscated version of case 1
        break;
      ...
      default: // obfuscated version of case 1
        break;
    }
  }

  if (hash == 809267)
    printf("You win!");

  return 0;
}
```

that if $O(P)$ is the obfuscated version of program $P : D \rightarrow R$, where $D, R \subset \{0,1\}^*$ are the input domain, respectively output range of the program, then $\forall i \in D : P(i) = O(P)(i)$. However, Barak et al. [26] also define an *approximate obfuscator* as a transformation for which the *functionality* property holds with high probability. Similarly to an approximate obfuscator, our obfuscation approach relaxes the *functionality* property, but does so in a different way. In our approach the *functionality* property only holds for the set of inputs that satisfy the *input invariants* (i.e. predicates over inputs), specified by the user to the obfuscation engine. For all other input values the behavior of the program is undefined. With this obfuscation approach, we are essentially extending the input domain and output range of a program, i.e. $O(P) : D' \rightarrow R'$, where $D \subseteq D'$ and $R \subseteq R'$. In fact one could imagine the extensions go even further, allowing $O(P)$ to *fail* in different ways than P, such as P crashing on bad inputs, while $O(P)$ entering an infinite loop on the same bad inputs, or producing the wrong results, etc. We believe this idea has very interesting implications for future implementations of obfuscation transformations.

An implementation of input invariants We have implemented this transformation on top of the *Virtualize* transformation of Tigress [55]. We picked *Virtualize* because of its high number of branches in the interpreter, however, note that this idea could be applied to the control-flow flattening as well. Intuitively, we want to make all of the branches of the interpreter dependent on a symbolic value, which in the case of bypassing license checks is the input argument. Therefore, we use the resulting value of the input invariant predicate as a key to encode the virtualized bytecode program, before sending it to the end-user. At

Listing 7.5: Point function program

```
1  int main(int argc, char* argv[]) {  // Virtualization bytecode:
2    char branch_cond = 1;             // a5 00 07
3
4    branch_cond &= argv[1][0] == '1'; // 87 00 00 02
5    branch_cond &= argv[1][1] == '2'; // 87 00 01 03
6    branch_cond &= argv[1][2] == '3'; // 87 00 02 04
7    branch_cond &= argv[1][3] == '4'; // 87 00 03 05
8    branch_cond &= argv[1][4] == '5'; // 87 00 04 06
9
10   if (branch_cond)                  // 1f 00 02
11     printf("You win!\n");           // 03 00
12   return 0;                         // 42 01
13 }
```

runtime, the input invariant predicate will be applied to the input given by the end-user and the result will be used to decode the bytecode, hence, causing all decoded bytecode values to be symbolic. This means that on each iteration of the interpreter the branching factor for the symbolic execution search tree is going to be equal to the number of instruction handlers of the interpreter. Moreover, if the attacker enters an input for which the invariant does not hold, then the functionality of the program is different from its intended functionality, i.e. it may crash due to the fact that there is no suitable instruction handler for the decoded bytecode instruction, or it may execute another instruction handler.

For clarity, we provide all of the steps of the *Virtualize* transformation, including the input invariants for the program in Listing 7.5. This program prints the message "You win!" on standard output if the first argument passed to this program is equal to "12345", similar to our license check programs from previous chapters. The virtualization transformation is applied to this program using the following steps and the result is illustrated in Listing 7.6:

1. Map variables, function parameters and constants to entries in a common `data` array, which represents the memory of the interpreter. This array is initialized on lines 3-4 in Listing 7.6. Its first position represents the `branch_cond` variable from Listing 7.5 and the following entries represent constants such as the return value, the ASCII codes of the characters from '1' to '5' and logical `true` encoded as 1.

2. Map all statements in a function to a new randomly chosen language, which represents the *instruction set architecture* (ISA) of the interpreter. In our example the ISA is defined by:
   - Variable assignment is encoded using 3 bytes, namely the opcode (`0xa5`) and the index of the left- and right-hand operands inside the `data` array.
   - Equality comparison, followed by applying the logical *AND* operation to between the result and another variable. Examples of such instructions are shown on lines 4-5 in Listing 7.5. Such an instruction is encoded using 4 bytes, namely the opcode (`0x87`), the variable to which the boolean value is assigned and the two other byte values which are compared for equality.

Listing 7.6: Point function program with virtualization and input invariants

```
1  int main(int argc, char* argv[]) {
2    char const *strings = "You win!\0";
3    unsigned char data[8] = {0,   // branch_cond var
4       0, 49, 50, 51, 52, 53, 1}; // constants
5    unsigned char code[30] = {0xa5^0x91, 0x00^0x91, 0x07^0x91, 0x87^0x91, 0x00^0x91,
6       0x00^0x91, 0x02^0x91, 0x87^0x91, 0x00^0x91, 0x01^0x91, 0x03^0x91, 0x87^0x91,
7       0x00^0x91, 0x02^0x91, 0x04^0x91, 0x87^0x91, 0x00^0x91, 0x03^0x91, 0x05^0x91,
8       0x87^0x91, 0x00^0x91, 0x04^0x91, 0x06^0x91, 0x1f^0x91, 0x00^0x91, 0x02^0x91,
9       0x03^0x91, 0x00^0x91, 0x42^0x91, 0x01^0x91};
10   unsigned char decode_var = str_hash(argv[1]);
11   for (int i = 0; i < 30; i++) {
12    code[i] ^= decode_var;
13   }
14   int vpc = 0;
15   while (1)
16   switch (code[vpc]) {
17     case 0xa5 : // variable assignment
18       data[code[vpc+1]] = data[code[vpc+2]];
19       vpc += 3;
20       break;
21     case 0x87 : // equality comparison plus and
22       data[code[vpc+1]] &=
23         (argv[1][code[vpc+2]] == data[code[vpc+3]]);
24       vpc += 4;
25       break;
26     case 0x1f : // if statement
27       vpc += (data[code[vpc+1]]) ? 0 : data[code[vpc+2]];
28       vpc += 3;
29       break;
30     case 0x03 : // printf string
31       printf("%s\n", strings + code[vpc+1]);
32       vpc += 2;
33       break;
34     case 0x42: // return
35       return data[code[vpc+1]];
36   }
37 }
```

- Conditional branch statements are encoded using 3 bytes, namely the opcode (`0x1f`), the boolean variable which is tested and the number of bytes to jump over if the variable is false.
- Printing a string on standard output is encoded using 2 bytes, namely the opcode (`0x03`) and the index of the string to be printed in the list of hard-coded strings of the function. In our example the list of hard-coded strings contains only one string and is defined on line 2 of Listing 7.6.
- The return instruction is encoded using 2 bytes, namely the opcode (`0x42`) and the value that should be returned by the program.

Now we can write virtualization bytecode corresponding to the C program in Figure 7.5, which is shown in the comments of the code from the same figure.

Listing 7.7: Hash function applied to strings

```
1 unsigned char str_hash(char* str) {
2   unsigned char hash = 0xAA;
3   unsigned int i = 0;
4   for(i = 0; *str != 0; str++, i++) {
5     hash ^= ((i & 1) == 0) ? ((hash <<  7) ^ (*str) * (hash >> 3)) :
6                              (~((hash << 11) + ((*str) ^ (hash >> 5))));
7   }
8   return hash;
9 }
```

Listing 7.8: Function from [220], which checks if first parameter has a value between the values of the second and third parameters.

```
1 int range_hash(int x, int lower, int upper) {
2     int p = ((x - lower) ^ ((x ^ lower) & ((x - lower) ^ x))) >> 31;
3     int q = ((upper - x) ^ ((upper ^ x) & ((upper - x) ^ upper))) >> 31;
4     return 1 & ~(p | q);
5 }
```

3. Generate a (random) hash or checksumming function to map an input invariant to an integer number. In our example we use the *str_hash* function from Listing 7.7, which maps a string to an unsigned char value. This function is appropriate when we use input invariants that require the input to be equal to a certain value. For input invariants where the input is expected to be an integer number between 10 and 25, the function would like the one from Listing 7.8. Note that these functions could be further obfuscated to raise the bar even further.

4. Use the hash or checksumming function to generate a key based on the input invariant. Encode each byte of the `code` array using this key. In our example applying the function from Listing 7.7 to "12345" returns value `0x91` and we *XOR* each byte of the code array with this value. The encoded bytecode is stored inside the `code` array, which is initialized on lines 5-9 in Listing 7.6.

5. Add bytecode decoder code to the function, based on the input value entered by the user. The decoder code can be seen on lines 10-13 in Listing 7.6, and it involves *XOR*-ing each byte of the code array with the result of applying the hash function to the user input. If the user input satisfies the invariants the bytecode is correctly decoded, otherwise the result is garbage bytecode, and the behavior of the program is undefined.

6. Create an interpreter for the previously generated ISA, which can execute the instructions in the `code` array using the `data` array as its memory. The input-output behavior of this execution must be the same as that of the original program. The interpreter can be seen on lines 15-36 of Listing 7.6. It consists of an infinite `while` loop, which has a `switch` statement inside. Each `case` section of the `switch` statement is

an opcode handler, i.e. each possible opcode in the bytecode program is processed by a dedicated part of the interpreter. The current instruction to be processed by the interpreter is indicated by an integer variable of the interpreter called the *virtual program counter* (VPC). The VPC is used to index the instructions in the `code` array and it is initialized with the offset of the first instruction in that array. In every instruction handler the operands of the current instruction are used to perform the operation(s) corresponding to this instruction. Afterwards, the VPC is set to the offset of the following bytecode instruction to be executed. This interpreter should be augmented with cases representing bogus opcodes for all possible byte values in order to increase the branching factor in the search tree associated to symbolic execution.

A user can specify the input invariants using: (1) the position of the argument in the list of arguments, (2) the type of the argument (integer or string) or its length and (3) the exact value or the interval of possible argument values. Note that different invariant types lead to keys with different cardinalities. The invariants with the highest cardinality keys are those that specify an integer or string argument with an exact value.

By using the invariant as a decoding key, we multiply the size of the search tree for symbolic execution by the cardinality of the range of possible key values (denoted C). However, if the number of different instruction opcodes (denoted O) is lower than C, then the branching factor of the search tree associated to symbolic execution is O. If we denote the length of a trace of a bytecode program – measured in number of random ISA instructions – as L, then the number of paths in the obfuscated program is equal to $\min(C, O)^L$.

As opposed to range dividers, – which cause an exponential path explosion when the original (un-obfuscated) program already contains a loop where the range divider predicate is inserted – our virtualization obfuscation based implementation of input invariants introduces a loop structure even if the original program does not contain one. Therefore, input invariants offer a higher degree of resilience against symbolic execution attacks, to a wider range of programs than range dividers.

Limitations of input invariants The effectiveness of this transformation against symbolic execution engines is higher than any other transformation we have employed in our case-study. To illustrate its effectiveness, we have chosen a program consisting of a single if-statement shown in Listing 7.5, because it is representative of the simplest possible code structure that one may want to protect against symbolic execution. We obfuscated this program using our modified *Virtualize* transformation with the invariant that the input is equal to 12345 and executed both the original program from Listing 7.5 and its obfuscated counterpart using KLEE. The point function from the program in Listing 7.5 was analyzed in approximately 500 milliseconds. While attempting to run the symbolic execution engine uninterrupted, similar to experiment 2 from Section 5.2.2, we resorted to stopping the analysis of the obfuscated program after it ran for 1 week. However, we note that the test suite that would find the path that prints "You win!" (the goal of experiment 5 in Section 5.2.3) was found in approximately 4980 seconds, which is still a slowdown by 4

orders of magnitude w.r.t. the unobfuscated counterpart. Contrast this with the smaller slowdown factors from Table 5.9.

7.4. Summary

In this chapter we have seen that obfuscation can affect search problems in two different ways, i.e.:

- Obfuscation can increase the branching factor and/or the depth of the goal state in the search tree, e.g. increasing the range of the license key implies a goal state at a greater depth in the search tree corresponding to random testing.
- Obfuscation can break the heuristic used by the search strategy, e.g. converting constant strings to procedures – that reconstruct the constant strings at runtime – breaks the heuristic used by the extraction of hard-coded standard C-strings.

Afterwards we have presented existing obfuscation transformations, specifically targeted towards raising the bar against attacks based on symbolic execution. These transformations are divided into three categories, named after the weakness of symbolic execution that they target, namely: path explosion, path divergence and complex constraints. Subsequently, we proposed two obfuscation transformations which result in a path explosion called: range dividers and input invariants. We chose to focus on obfuscation transformations that cause a path explosion, because in Chapter 5 we observed that many existing obfuscation transformations that increase cyclomatic complexity of the program, do not cause a path explosion. The two proposed obfuscation transformations are tunable according to the application to be protected and cause an exponential increase in the number of paths in the search tree corresponding to symbolic execution.

Part III.

Related Work and Conclusion

8. Related Work

This chapter presents related work in the sub-field of obfuscation strength evaluation and alternative solutions to software obfuscation and diversity for the purpose of software protection. Parts of this chapter have been published in peer-reviewed publications [21, 18, 17, 112] co-authored by the author of this thesis.

The main contribution of this thesis is related to the characterization of obfuscation strength. Therefore, we describe different existing approaches to this problem. Moreover, we also present alternative and complementary approaches to software obfuscation and diversity. Figure 8.1 shows a classification of protections against MATE attacks proposed by Collberg et al. [58]. We will briefly discuss about each of these protection classes in this chapter. The *obfuscation* class from Figure 8.1 can be generalized to any software-only protection that does not rely on trusted entities. Consequently, we also present a few complementary approaches to obfuscation that fit into this class. Complementary approaches include those technical protections that should be mixed with software obfuscation and code diversity to raise the bar against attacker.

8.1. Characterizing Obfuscation Strength

Since the obfuscation taxonomy of Collberg *et al.* [58], there have been several works that tackle the problem of characterizing or quantifying the strength of obfuscation from a variety of angles. In this section we classify them in three main categories, namely (1) formal approaches, (2) empirical approaches based on human-assisted MATE attacks and (3) empirical approaches based on automated MATE attacks. In the following we present representative works from each of these classes and we contrast them to this thesis.

8.1.1. Formal Approaches

Dalla Preda [64] models attacks against obfuscation transformations as abstract domains expressing certain properties of program behaviors. Since obfuscation transformations are characterized by the most concrete preserved property, the complete lattice of abstract domains allows comparing obfuscation transformations with respect to their potency against various attackers. Therefore, an obfuscation transformation is either effective against an attacker or not, regardless of the difference in effort needed to deobfuscate

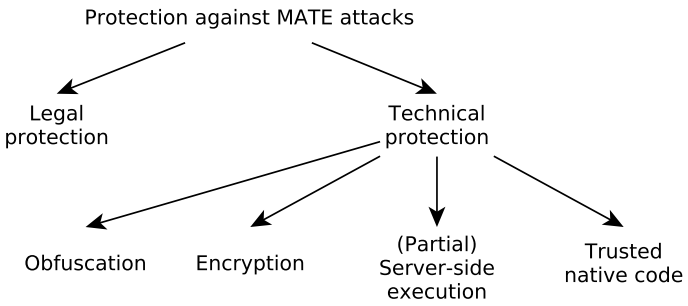


Figure 8.1.: Classification of protections against MATE attacks proposed in [58].

programs obfuscated with different transformations. Our work provides a more fine grained characterization of the resilience of obfuscation transformations w.r.t. to the effort required by the attacker to deobfuscate a program.

Pucella and Schneider [174] investigate the effectiveness of defenses based on software diversity, in the context of memory safety. Their main result is to characterize such defenses as to be probabilistically equivalent to strong typing which would guarantee memory safety for buffers, thus reducing the security of the defense mechanism to the strength of strong typing. In particular, they analyze *address obfuscation* [32], a defense mechanism against memory corruption attacks, that uses a secret key to randomize the offsets of code and data in heap memory. Their idea is to treat address obfuscation as a probabilistic type checker, which has a certain probability p of crashing the program when a buffer overflow occurs. As opposed to Pucella and Schneider [174], this thesis does not consider attacks that exploit memory safety vulnerabilities, but automated MATE attacks. Moreover, we provide a framework to reduce any automated MATE attack to search problems, whose expected cost can be estimated using characteristics of the program under attack or any artifact (e.g. instruction execution traces), generated by that program.

Ganesh et al. [88] propose the concept of *attack resistance*, which takes a crypto-like approach towards quantifying the strength of defenses based on software diversity against known attacks, e.g. code injection. As opposed to our framework which covers all MATE attackers, *attack resistance* only focuses on code injection attacks, which exploit buffer overflow vulnerabilities. Moreover, *attack resistance* quantifies the strength of a defense mechanism (e.g. ISR [126]), by indicating the probability of a successful attack, while we compute the average effort for an attack measured in time (e.g. CPU cycles) or memory (e.g. MBs).

8.1.2. User Studies

Sutherland et al. [199] and Ceccato et al. [43, 44] characterize the strength of obfuscation transformations by potency against human-assisted attacks, e.g. step-by-step debugging. This involves user studies where test subjects are asked to perform some tasks (e.g. bypass a

check or recover information) on code obfuscated with a limited number of transformations. Such user-studies are inherently biased by small sets of test subjects. Moreover, test subjects are generally bachelor or master students of computer science, which are seldom experienced in reverse engineering obfuscated code. Nevertheless, such works are important for measuring potency against human-assisted attacks; our work is complementary to this approach.

Sutherland et al. [199] perform a user study, where they find that the ability/knowledge of the MATE attacker is significantly correlated with the success rate of the subjects. On the other hand they also present empirical results showing that the success rate of the human-assisted MATE attacks are not strongly correlated with the number of LOC, cyclomatic complexity or with any Halstead metrics. This result is similar to our finding from Chapter 6, however, we have only focused on automated MATE attacks using symbolic execution, not on human-assisted attacks.

Ceccato et al. [43] confirm the results of Sutherland et al. [199] for un-obfuscated programs, however, they find that by obfuscating these programs, the gap between the success rate of knowledgeable and less knowledgeable subjects decreases. This means that even knowledgeable MATE attackers must spend a large amount of effort when dealing with obfuscated code. In another work Ceccato et al. [44] investigate the relative strength of two different obfuscation transformations, namely *scrambling identifier names* and *opaque predicates*. They find that *scrambling identifier names* poses more challenges for human-assisted attacks than *opaque predicates*. This holds in the context where identifiers in the original program have a proper semantic meaning (in English). This finding emphasizes the difference between human-assisted MATE attacks – which is the focus of user studies – and automated MATE attacks – which is the focus of this thesis – because *scrambling identifier names* has no effect on automated attacks.

8.1.3. Code Metrics Based Approaches

Udupa et al. [209] propose using the edit distance between control flow graphs of the original code and deobfuscated code. Mohsen and Pinto [155] propose using Kolmogorov complexity. However, they do not attempt to predict the effort needed for deobfuscation, which has been identified as a gap in this field [199]. In this thesis we focus on predicting the effort needed by a deobfuscation attack.

Karnick et al. [124] proposed to measure the quality of Java obfuscators by summing up potency and resilience and subtracting cost of memory consumption, file storage size and execution time from the sum. They measure potency with a subset of the features proposed by Collberg et al. [58]. They measure resilience by using concrete implementations of deobfuscators, measuring whether they were successful or if they encountered errors and averaging the measurements across the total number of deobfuscators. We acknowledge that using multiple concrete implementations of a deobfuscation attack (e.g. disassembly, CFG simplification) is important to weed out any issues specific to a particular implementation. In this work we aim to provide a more fine-grained measure of deobfuscation effort,

instead of a categorical classification such as *succeeded* or *failed* for each deobfuscation attack implementation, as done in [124]. Moreover, we also predict this fine-grained effort.

Anckaert et al. [7] propose applying concrete software complexity metrics on four program properties (i.e. instructions, control flow, data flow and data), to measure resilience. Several of the metrics indicated in [7] overlap with those proposed by Collberg et al. [58], however, other metrics are included such as: number of operands of an instruction, the number of different values of these operands, knot count [223], number of live values, size of points-to sets, number of def-use pairs [202], distance between def-use pairs in traces. Similarly to our work, Anckaert et al. [7] measure resilience of different obfuscating transformations against concrete implementations of deobfuscation attacks. Nonetheless, they apply deobfuscation attacks which are specific to different obfuscating transformations, while we use a general deobfuscation attack (based on symbolic execution) on all obfuscating transformations. Moreover, they disregard the effort needed for deobfuscation and measure the effect of different obfuscating transformations on software complexity metrics and the subsequent effect of deobfuscation on these metrics. In this thesis we not only measure but also predict the effort needed to run a successful deobfuscation attack. Moreover, the experiments from Section 6.2 indicate that SAT features – that are most important for slowing down symbolic execution – are not affected by applying control-flow obfuscation transformations such as opaque predicates, even though code complexity metrics are increased.

Vissagio et al. [213] propose using a combination of static code metrics (i.e. n-gram, entropy and word size) in order to detect if a certain JavaScript program is obfuscated or not. In the process they also compare which of the four different obfuscation transformations are more resilient to this detection approach by comparing the p-values of different statistical tests (i.e. Mann-Whithney and Kolmogorov-Smirnov). We see their work as complementary to ours, however, as opposed to our approach, it does not give an indication of the effort needed by the attacks.

Wang et al. [217] also use attack nets in order to quantify the cost of MATE attacks. As opposed to our work, they assume a range of six discrete values for the cost of a transition, i.e. $\{1, 5, 10, 15, 20, 25\}$ where higher values indicate a higher cost of the transition. In this thesis, we focus on a fine grained quantification of transitions.

Wu et al. [224] propose using a linear regression model over a fixed set of features, for measuring the potency of obfuscating transformations. In contrast to our work, they do not provide any evaluation of their approach. They suggest obtaining the ground truth for training and testing a linear regression model, from security experts who manually deobfuscate the obfuscated programs and indicate the effort required for each program, which is far more expensive compared to our approach of using automated attacks. We obtain our ground truth by running an automated attack and recording the effort (measured in execution time), needed to deobfuscate programs. Moreover, we also propose a way of selecting which features to use for building a regression model.

In sum, Collberg's taxonomy [58] proposes evaluating obfuscation using four dimensions. Most of the related work focuses on simply *measuring* potency, resilience and cost. Wu

et al. [224] discuss estimating potency. Zhuang and Freiling [233] propose using a naive Bayes algorithm to estimate the optimal sequence of obfuscating transformations, from a performance point of view. Kanzaki et al. [122] propose code artificiality as a measure to estimate stealth. However, there is a gap in estimating resilience, which we fill in this work.

8.2. Alternatives to Diverse Obfuscation

As indicated in Figure 8.1, there are several ways in which one could protect against MATE attacks. Since in this thesis we focus on technical protection via obfuscation, we present the other types of protections in the remainder of this chapter.

8.2.1. Encryption via Trusted Hardware

Software protection via encryption is usually enabled by trusted hardware, also called *trusted computing*. Intel has released a hardware based technology [5], known as *Software Guard eXtension* (SGX), which enables software developers to protect the confidentiality of their applications' code via protected execution areas called *enclaves*. Dewan et al. [71] also use a trusted hypervisor to protect the sensitive memory of programs against unauthorized access by leveraging trusted hardware. Feng et al. [81] propose performing randomly-timed stealthy measurements which can be validated locally, using Intel's Active Management Technology [113]. These approaches provide high security guarantees. However, they require trusted hardware to be available and the installation of a hypervisor. Software developers of popular software (e.g. web browsers), generally do not want to restrict their user base by imposing such requirements.

8.2.2. Server-Side Execution

Tamper protection via communication with trusted servers is employed in massive multi-player online games (MMOGs) to detect cheating. Anti-cheat software such as PunkBuster [78], Valve Anti-Cheat (VAC) [211], Fides [121] and Warden [107] perform client-side computation, which are validated by a trusted server.

Martignoni et al. [145] and Seshandri et al. [187] propose establishing a *trusted computing base* to achieve verifiable code execution on a remote un-trusted system. The trusted computing base in the two methods is established using a verification function. The verification function is composed of three components: (i) a checksum function, (ii) a send function, and (iii) a checksum function. However, the main difference between the two methods is the checksum function. In the work of Martignoni et al. [145] generates a new checksum function each time and sends it encrypted to the un-trusted system. In the work of Seshandri et al. [187], the checksum function is known a priori and the challenge issued by the dispatcher consists in a seed that initializes this function. Since the remote component in both methods knows precisely in which execution environment the function

must be executed and knows the hardware characteristics of the un-trusted system, it can compute the expected checksum value and can estimate the amount of time that will be required by the un-trusted system to decrypt and execute the function, and to send back the result. Since Intel x86 architecture, the architecture for which the approach of Seshandri et al. [187], was developed, is full of subtle details, researchers have found ways to circumvent the remote component. Also, a limitation of the approach of Martignoni et al. [145], is the impossibility to bootstrap a tamper-proof environment on simultaneous multi threading (SMT) or simultaneous multi processing (SMP) systems. On such systems, the attacker can use the secondary computational resources (parallel threads for example) to forge checksums or to regain control of the execution after attestation.

Jakobsson and Johansson [119] propose a similar technique for detecting malware on mobile devices. Collberg et al. [56] propose tamper protection by pushing continuous updates from a trusted server to the client, which force the attacker to repeat reverse engineering and patching on each update. One disadvantage of these protection techniques is their dependence on external trusted servers. This dependence may cause a denial-of-service to end-users of the protected software applications which are also meant to be used offline, in case Internet connectivity is unavailable. Therefore in this thesis we focus on solutions that operate locally, i.e., without dependence on a trusted server.

8.2.3. Code Tamper-detection and Tamper-proofing

Code tamper-detection and tamper-proofing are complementary techniques to software diversity and obfuscation and they aim to detect, respectively prevent unauthorized modifications of a program's code. However, these techniques are not generally stealthy, and hence they should be combined with diverse obfuscation in order to hamper MATE attackers from disabling such mechanisms.

Chang and Atallah [45] propose building a network of code regions, where a region can be a block of user code, a checker, or a responder. In this method checkers check each other in addition to user code by comparing a known checksum of piece of code to runtime checksum of the same code. If the checker has discovered that a region has been tampered with, a responder will replace the tampered region with a copy stored elsewhere. An important aspect of this algorithm is that it is not enough for checkers to check just the code, they must check each other as well. If checkers are not checked, they are easy to remove. Horne et al. [110] build on top of [45], by hiding the expected (precomputed checksum) value which is easy to identify, because of its randomness. The idea is to construct the checksum function such that unless the code has been tampered with, the function always checksums to a known number (usually zero). Having this function allows to insert an empty slot within the region under protection, and later give this slot a value that makes the region checksum to zero. The technique of Horne et al. [110] randomly places large numbers of checkers all over the program, but makes sure that every region of code is covered by multiple checkers. To minimize pattern-matching attacks, this method describes how to generate a large number of variants of lightweight checksum functions. The disadvantage

of the *code introspection* approach used by both [45] and [110] is its stealthiness, because code that reads itself is seldom used for other purposes.

Chen et al. [46] propose an idea called *oblivious hashing*, where the checksum value is computed over the *execution trace* rather than the static code. The checksum can be computed by inserting instructions that monitor changes to variables and the execution of instructions. A problem with automating this technique, is that it is hard to predict what side effects a function might have. It might destroy valuable global data or allocate extraneous dynamic memory that will never be properly freed. Furthermore, there is a problem with non-deterministic functions that depend on the time of day, network traffic, thread scheduling, and so on, because they do not have a fixed output that can be checked. This technique also faces the issue of automatically generating challenge data (test inputs) that most of the code of a function. The approach by Ibrahim and Banescu [112] implements a variant of oblivious hashing therefore it also suffers from the same disadvantages. However, we address the last issue by proposing the use of symbolic execution in order to generate the challenge data.

Jacob et al. [117] propose an approach which depends on a unique property of the x86 instruction set architecture (ISA). The x86 ISA has a *variable instruction length* (1-15 bytes) with no alignment, this means instructions can start at any offset in the code. This results in the possibility of having overlapping or even nested instructions. So the basic idea will be that when a block is executed it computes a checksum of another block. For the purpose of protecting the code, we need two blocks to share instruction bytes. Having two blocks to share instruction bytes, can be achieved by interleaving the instructions and inserting jumps to maintain semantics. The advantage in this technique is that the code checksumming computations will not require reading the code explicitly. The disadvantage is mainly the performance overhead of the added instructions. Jacob et al.[117] report that the protected binary can be up to three times slower than the original. Even though this overhead may be acceptable in many circumstances, this technique cannot be applied to programs that execute on the Common Language Runtime such as programs written in C#.

Cappaert et al. [41] propose a technique that hinders both code analysis and tampering attacks simultaneously through code encryption. During run-time, code decryption can be done at a chosen granularity (e.g. one function at a time), when that part of code is needed at run-time. This technique performs integrity-checking of the code by using it to compute the keys for decryption and encryption. The basic idea is using the checksum value of a function, as the decryption key of another function. The advantage of this technique is that the encryption key is computed at run time, which means the key is not hard-coded in the binary and therefore hard to find through static analysis. The disadvantage of this technique is the run-time overhead as well as the its stealth.

8.3. Summary

On the one hand, formal approaches to characterizing strength of software protection either provide coarse grained quantification (e.g. resistant or not resistant against an attack) or they focus on remote attacks. User studies generally focus on measuring the potency of human-assisted MATE attacks. We focus on automated MATE attacks.

On the other hand, approaches based on code metrics focus only on measuring: potency, resilience and cost, which also considers automated MATE attacks. We do not only measure the resilience against automated MATE attacks, but also provide a search model that can be used to reason about the characteristics which are most relevant for such attacks. Moreover, we use the identified characteristics to predict the time needed by a symbolic execution to successfully attack a given program. Finally, we also provide solutions in the form of novel obfuscation transformations for raising the bar against such symbolic execution attacks.

Alternatives to obfuscation and software diversity that rely on trusted hardware or an external trusted server, may offer stronger security guarantees than software obfuscation, however, they also involve higher deployment costs. Other alternatives based on tamper-detection and tamper-proofing are often mixed together with obfuscation and software diversity in order to improve their stealthiness.

9. Conclusions

This chapter first presents a summary of what has been done throughout the chapters of this thesis. Subsequently, we state the results of the thesis and the lessons learned during the development of this work. Afterwards, we discuss limitations and avenues for future work.

This thesis presents a general approach towards characterizing the strength of obfuscation against any automated Man-At-The-End (MATE) attack in the context of software diversity. Even though the framework is not specific to practical obfuscation transformations, we have only applied it to such transformations, because cryptographically secure obfuscation is still far from being practical (see Chapter 2). Since there are dozens of practical obfuscation transformations proposed in the literature (see Chapter 3), it is difficult for practitioners to choose which transformations to use, even if they know the capabilities of the MATE attacker. Therefore, in Chapter 4 we propose formalizing different steps of an automated MATE attack as search problems. This formalization facilitates reasoning about the software features which are most relevant for estimating the effort of such attacks, because this effort is proportional to the search cost. This estimation enables practitioners to select those obfuscation transformations which affect the software features that increase the effort of the attack.

Our framework is applicable to any automated MATE attack. However, since it is not feasible to do an in-depth study of all existing MATE attacks in the span of one doctoral thesis, we wanted to find common step for several automated MATE attacks and perform an in-depth case-study on it. We identified a common subgoal for the majority of dynamic MATE attacks, which include, but are not limited to the following three attacker goals:

- Simplifying the CFG of an obfuscated program, as proposed by Yadegari et al. [228].
- Identifying and disabling the integrity checks hidden inside of an obfuscated program, as proposed by Qiu et al. [175].
- Bypassing the license check(s) in obfuscated software with premium features, as proposed by Banescu et al. [21].

This common subgoal is *test case generation* for the purpose of executing all the code of the obfuscated program and finding specific paths in an obfuscated program (see Chapter 5). This common subgoal allows us to set a limit to the number of obfuscation transformations

that should be applied to a program in order to defend against the previously enumerated attacker goals (see Chapter 5).

Since test case generators involve *searching* for different test cases, we can formalize state-of-the-art test case generators as search problems using our framework from Chapter 4. For this purpose we pick test case generators based on symbolic execution and perform different case-studies using manually written and automatically generated programs. The reasons why we picked symbolic execution is that it represents a state of the art technique for white-box test case generation. As opposed to black-box test case generation – which is not affected by code obfuscation – symbolic execution is affected by code obfuscation transformations, which are the focus of this thesis. In the case-studies we measured the search cost of symbolic execution for thousands of programs obfuscated with various practical code obfuscation transformations. The results confirmed that the software features which we identified by formalizing symbolic execution as a search problem, are actually relevant to characterize the effort of this automated MATE attack (see Chapter 5).

The software features that were identified to be relevant for characterizing the effort of an automated symbolic execution attack, could be measured using dozens of metrics. Since it is unclear which metrics a software developer should focus on when aiming to produce the highest increase in effort for symbolic execution attacks, we proposed an approach to prioritize these metrics by using regression models to predict the attacker effort (see Chapter 6). This way we determined which features are the most important when it comes to predicting the time needed for such attacks.

Finally, we have leveraged the information regarding the most relevant software features for symbolic execution to propose novel obfuscation techniques, which raise the bar against such automated MATE attacks (see Chapter 7). An implementation of one of the obfuscation transformations we proposed, was able to increase the effort of symbolic execution by four orders of magnitude w.r.t. applying the same attack on a program that was not obfuscated using our technique.

Overall Conclusion Altogether, the chapters of this thesis answer all the research questions posted in Section 1.5. This indicates reaching the goal of this thesis (see Section 1.4), i.e. having a framework that can characterize the strength of obfuscation and aid practitioners in choosing and/or developing new obfuscation transformations against all automated MATE attacks.

9.1. Results and Lessons Learned

In this section we highlight some of the most interesting results and discuss the lessons learned during the development of this thesis.

Obfuscation strength is proportional to effort of the best known automated MATE attack. The strength of obfuscation can be quantified by estimating the effort of the best

known automated MATE attack. There are multiple known automated MATE attacks that can achieve a certain attacker goal. Depending on the obfuscation transformations applied to a program, an attacker will choose to apply the cheapest – in terms of time and/or memory – automated MATE attack. If the attacker does not know which of the attacks is the cheapest, s/he can run all known attacks in parallel and chose the best/cheapest attack.

All automated MATE attacks can be formalized as one or more search problems. The key insight of this thesis is that all automated MATE attack consist of one or more steps. At least one of these steps is a search problem, whose average case complexity can be estimated using characteristics of the data structure used by the search algorithm. Moreover, many automated MATE attacks have common steps. One example of a common step identified in this thesis is automated test case generation.

Several control-flow obfuscation transformations are weak against symbolic execution. Our empirical case-studies indicate that for the datasets of programs used in this thesis, a subset of the existing obfuscation transformations are weak against white-box test case generation via symbolic execution, when applied once, individually. This subset includes popular obfuscation transformations described in Chapter 3, such as:

- Encoding (converting) constant literals to procedures, which compute these constant values dynamically.
- Adding bogus control flow and dead code via opaque predicates, which are expressions whose values are invariant during runtime.
- Flattening the control-flow of an application into a giant switch statement.
- Virtualization obfuscation, which maps the original program to a randomly generated ISA and an interpreter for that ISA.

The rationale behind this finding is that such obfuscation transformations do not add additional execution paths dependent on symbolic variable, even though they do add numerous branches to the program. The most resilient obfuscation transformation w.r.t. symbolic execution attacks has proved to be arithmetic encoding, because it is the transformation that increases the SAT features of a program to the highest extent. However, we have also observed that virtualization and control-flow flattening do increase the time needed by symbolic execution due to other software features. Moreover, we have observed that by combining multiple obfuscation transformations the cost of symbolic execution is increased significantly.

SAT features are more important than cyclomatic complexity, w.r.t. characterizing the effort of symbolic execution attacks. For programs for which our generated dataset is representative, features such as the complexity of path constraints (measured via SAT

features), are more important than cyclomatic complexity, size of the program, number of conditional operations, etc. This insight is different from other works that focus on characterizing the strength of obfuscation, which often use cyclomatic complexity and program size as metrics for obfuscation strength. The rationale behind this observation is that SAT features capture the complexity from instructions involving only symbolic variables, and do not include complexity added by other instructions – like other code metrics do – which do not significantly impact the time needed to symbolically execute a program.

The effort needed by an automated MATE attack can be estimated using regression based on the software features identified using our search model. Using the software features identified using our search model, we were able to build regression models that could estimate the effort (i.e. time) needed by a symbolic execution attack. With a median error of 4% our best regression model can accurately predict the time it takes to deobfuscate a program using a symbolic execution based attack, for programs in our dataset. Moreover, we have also obtained encouraging results with realistic hash functions such as MD5 and SHA instances used in SAT competitions. Such a prediction of the effort for an attack on a certain application is important for practitioners, who can thus easily decide if they must protect their applications using additional obfuscation transformations, without actually needing to invest the effort to run the attack themselves. Note that such an attack could cost days or weeks of computation time and delay time to market.

9.2. Limitations

The most important limitation of this thesis is the use of small, artificially created programs, as opposed to performing a case-study using real-world programs, namely programs containing calls to functions from third party libraries, OS API functions, etc. The main reason behind this limitation is due to the fact the current symbolic execution engines do not scale for most large (real-world) programs. Symbolic execution has several limitations when applied to large software programs (see Section 7.2). Applying symbolic execution with a small cutoff value of 1 hour – as we did for the thousands of small programs in our case-studies – to a large program, would lead to a timeout of the SAT-/SMT-solver or the symbolic execution engine itself. Such a timeout would mean an unsuccessful attack in the context of our case-studies. However, symbolic execution is currently a highly active field of research and has been successfully applied for finding bugs in Microsoft Windows 7 [95] and it is being used by several teams in the DARPA Cyber Grand Challenge for automated exploit generation [11]. Moreover, several deobfuscation techniques rely on symbolic execution [185]. Furthermore, obfuscation is often applied only to parts of software to minimize performance impact [57], hence attackers may isolate and symbolically analyze smaller parts of the software [3]. Nevertheless, we cannot claim that the results from our case-studies from Chapter 5 and Chapter 6 generalize to all programs. Our results also

inherit all the limitations and bugs inside tools used in our experiments (e.g. Tigres, KLEE, angr, etc.).

The search model – which is the main insight of this thesis – can be applied to any other automated MATE attack, which may scale to real-world programs as well. Such attacks include those executed by modern fuzzers (e.g. AFL [231]), which were not investigated in this thesis. Although, such attacks are a main direction of future work.

Our framework of characterizing obfuscation strength based on the effort needed by the best known automated MATE attack, does not take into account unknown (e.g. future) attacks. This means that our framework can only provide an upper bound on the lower bound of the attacker effort, not a lower bound, which would be ideal for establishing formal security guarantees. However, we note that the strength of standard cryptographic ciphers (e.g. AES, RSA, ECC, etc.) is also quantified in a similar manner [74], namely using the strength of the best known attacks against such ciphers.

9.3. Future Work

One direction of future work is to instantiate our search based framework and perform case-studies using other automate MATE attacks. In this thesis we have focused only on attacks based on symbolic execution, because they require no manual adjustments when applied against several different popular code obfuscation transformations. However, in order to bring additional empirical support for our hypothesis, all of the attacks described in Chapter 4 should be investigated in-depth, at least to the same extent as we have investigated symbolic execution in Chapters 5, 6 and 7.

In future work we also plan to use datasets consisting of real-world programs and additional obfuscation tools. We believe that obtaining representative datasets of programs would also be of paramount importance for benchmarking both new and existing obfuscation and deobfuscation techniques. Therefore, we believe the research area of benchmark (program) generation needs much more work, since it could be a driving factor for the field of software protection.

Another avenue for future work is to employ other machine learning techniques in order to derive better prediction models for deobfuscation attacks. An interesting idea in this direction is deriving software features relevant for increasing automated MATE attack effort using deep neural networks. However, such a task would also require a set of representative un-obfuscated programs, which stresses the importance of future work in the direction of benchmark generation.

Bibliography

[1] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. *Journal of Network and Computer Applications*, 48:44–57, 2015.

[2] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[3] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381. Springer, 2008.

[4] P. Ananth, D. Gupta, Y. Ishai, and A. Sahai. Optimizing obfuscation: Avoiding barrington's theorem. *IACR Cryptology ePrint Archive*, 2014:222, 2014.

[5] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.

[6] Bertrand Anckaert, Mariusz H Jakubowski, Ramarathnam Venkatesan, and Chit Wei Saw. Runtime protection via dataflow flattening. In *2009 Third International Conference on Emerging Security Information, Systems and Technologies*, pages 242–248. IEEE, 2009.

[7] 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.

[8] S. Arora and B. Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[9] Arxan. Obfuscation. `https://www.arxan.com/technology/obfuscation/`. Accessed: 2017-03-03.

[10] David Aucsmith. Tamper resistant software: An implementation. In *Information Hiding*, pages 317–333, 1996.

[11] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[12] David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[13] Lee Badger, Larry D'Anna, Doug Kilpatrick, Brian Matt, Andrew Reisse, and Tom Van Vleck. Self-protecting mobile agents obfuscation techniques evaluation report. *Network Associates Laboratories, Report*, pages 01–036, 2002.

[14] S. Banescu, M. Ochoa, A. Pretschner, and N. Kunze. Benchmarking indistinguishability obfuscation - a candidate implementation. In *Proc. of 7th International Symposium on ESSoS*, number 8978 in LNCS, 2015.

[15] Sebastian Banescu. Obfuscation Benchmarks. `https://github.com/tum-i22/obfuscation-benchmarks`. Accessed: 2017-03-12.

[16] Sebastian Banescu. Cache timing attacks. `http://www.academia.edu/download/31092493/report.pdf`, 2011. Accessed: 2017-03-30.

[17] Sebastian Banescu, Mohsen Ahmadvand, Alexander Pretschner, Robert Shield, and Chris Hamilton. Detecting patching of executables without system calls. In *Proceedings of the Conference on Data and Application Security and Privacy*, CODASPY '17, 2017.

[18] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proc. of 2016 Annual Computer Security Applications Conference*. ACM, 2016.

[19] Sebastian Banescu, Christian Collberg, and Alexander Pretschner. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[20] Sebastian Banescu, Ciprian Lucaci, Benjamin Krämer, and Alexander Pretschner. Vot4cs: A virtualization obfuscation tool for c. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, pages 39–49. ACM, 2016.

[21] Sebastian Banescu, Martín Ochoa, and Alexander Pretschner. A framework for measuring software obfuscation resilience against automated attacks. In *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, pages 45–51. IEEE, 2015.

[22] Sebastian Banescu and Alexander Pretschner. A tutorial on software obfuscation. In *Advances in Computers*. Elsevier, 2018.

[23] Sebastian Banescu, Alexander Pretschner, Dominic Battré, Stéfano Cazzulani, Robert Shield, and Greg Thompson. Software-based protection against changeware. In *Proceedings of the Conference on Data and Application Security and Privacy*, CODASPY '15, pages 231–242, 2015.

[24] Sebastian Banescu, Tobias Wuchner, Aleieldin Salem, Marius Guggenmos, Alexander Pretschner, et al. A framework for empirical evaluation of malware detection resilience against behavior obfuscation. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 40–47. IEEE, 2015.

[25] Boaz Barak. Hopes, fears, and software obfuscation. *Communications of the ACM*, 59(3):88–96, 2016.

[26] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *Advances in Cryptology CRYPTO 2001*, pages 1–18. Springer, 2001.

[27] Eda Sevim Barlak. Feature selection using genetic algorithms. 2007.

[28] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[29] D. A. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc1. In *Proc. of the 18th Annual ACM Symp. on Theory of Computing*, STOC '86, pages 1–5, New York, NY, USA, 1986. ACM.

[30] Cataldo Basile, Stefano Di Carlo, Thomas Herlea, Verizon Business, Jasvir Nagra, and Brecht Wyseur. Towards a formal model for software tamper resistance. In *Second International Workshop on Remote Entrusting (ReTtust 2009)*, volume 16.

[31] Benoit Baudry and Martin Monperrus. The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Computing Surveys (CSUR)*, 48(1):16, 2015.

[32] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the USENIX Security Symposium*, 2003.

[33] Sandeep Bhatkar and R. Sekar. Data space randomization. In Diego Zamboni, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, number 5137 in Lecture Notes in Computer Science, pages 1–22. Springer Berlin Heidelberg, January 2008.

[34] Abdulazeez S Boujarwah and Kassem Saleh. Compiler test case generation methods: a survey and assessment. *Information and software technology*, 39(9):617–625, 1997.

[35] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[36] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[37] Julien Bringer, Herve Chabanne, and Emmanuelle Dottax. White box cryptography: Another attempt. *located at, last visited on Jul*, 22(2011):14, 2006.

[38] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, volume 36 of *Advances in Information Security*, pages 65–88. Springer US, 2008.

[39] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[40] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM. 00041.

[41] Jan Cappaert, Bart Preneel, Bertrand Anckaert, Matias Madou, and Koen De Bosschere. Towards tamper resistant code encryption: Practice and experience. In *Information Security Practice and Experience*, pages 86–100. Springer, 2008.

[42] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163. Springer, 2008.

[43] 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 *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 178–187. IEEE, 2009.

[44] Mariano Ceccato, Massimiliano Di Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19(4):1040–1074, February 2013.

[45] Hoi Chang and Mikhail J Atallah. Protecting software code by guards. In *Security and privacy in digital rights management*, pages 160–175. Springer, 2001.

[46] Yuqun Chen, Ramarathnam Venkatesan, Matthew Cary, Ruoming Pang, Saurabh Sinha, and Mariusz H Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *International Workshop on Information Hiding*, pages 400–414. Springer, 2002.

[47] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[48] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. ASPLOS XVI, pages 265–278, New York, NY, USA, 2011. ACM.

[49] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C. Van Oorschot. A white-box DES implementation for DRM applications. In *Digital Rights Management*, pages 1–15. Springer, 2003.

[50] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C. Van Oorschot. White-box cryptography and an AES implementation. In *Selected Areas in Cryptography*, number 2595 in LNCS, pages 250–270. Springer Berlin Heidelberg, January 2003.

[51] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *International Conference on Information Security*, pages 144–155. Springer, 2001.

[52] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[53] F. B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, October 1993.

[54] C. Collberg, J. Davidson, R. Giacobazzi, Y. X. Gu, A. Herzberg, and F. Wang. Toward digital asset protection. *Intelligent Systems, IEEE*, 26(6):8–13, 2011.

[55] Christian Collberg. The Tigress C Diversifier/Obfuscator. `http://tigress.cs.arizona.edu/`. Accessed: 2016-11-29.

[56] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 319–328, New York, NY, USA, 2012. ACM.

[57] Christian Collberg and Jasvir Nagra. Surreptitious software. *Upper Saddle River, NJ: Addision-Wesley Professional*, 2010.

[58] 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.

[59] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

[60] 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, New York, NY, USA, 2011. ACM.

[61] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[62] 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.

[63] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard TM: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.

[64] Mila Dalla Preda. *Code obfuscation and malware detection by abstract interpretation*. PhD thesis, University of Verona, 2007.

[65] Mila Dalla Preda and Roberto Giacobazzi. Control code obfuscation by abstract interpretation. In *Third IEEE International Conference on Software Engineering and Formal Methods.*, pages 301–310. IEEE, 2005.

[66] George B Dantzig, Alex Orden, Philip Wolfe, et al. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

[67] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[68] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[69] Bjorn De Sutter, Bertrand Anckaert, Jens Geiregat, Dominique Chanet, and Koen De Bosschere. Instruction set limitation in support of software diversity. In *International Conference on Information Security and Cryptology*, pages 152–165. Springer, 2008.

[70] Frédéric Delbos and Jean Charles Gilbert. *Global linear convergence of an augmented Lagrangian algorithm for solving convex quadratic optimization problems*. PhD thesis, INRIA, 2003.

[71] Prashant Dewan, David Durham, Hormuzd Khosravi, Men Long, and Gayathri Nagabhushan. A hypervisor-based system for protecting software runtime memory and persistent storage. In *Proceedings of the 2008 Spring simulation multiconference*, pages 828–835. Society for Computer Simulation International, 2008.

[72] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, SFCS '81, pages 350–357, Washington, DC, USA, 1981. IEEE Computer Society.

[73] Chris Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.

[74] II ECRYPT. Yearly report on algorithms and keysizes (2011-2012), d. spa. 20 rev. 1.0. `http://www.ecrypt.eu.org/ecrypt2/documents/D.SPA.20.pdf`. Accessed: 2017-03-30.

[75] C. Edwards. Researchers probe security through obscurity. *Communications of the ACM*, 57(8):11–13, 2014.

[76] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[77] Rakan El-Khalil and Angelos D Keromytis. Hydan: Hiding information in program binaries. In *International Conference on Information and Communications Security*, pages 187–199. Springer, 2004.

[78] Evenbalance. PunkBuster — Online Countermeasures, 2015. `http://www.evenbalance.com/pbsetup.php`, [Online; accessed 20-September-2016].

[79] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating mba-based obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, pages 27–38. ACM, 2016.

[80] Paolo Falcarin, Christian Collberg, Mikhail Atallah, and Mariusz Jakubowski. Guest editors' introduction: Software protection. *Software, IEEE*, 28(2):24–27, 2011.

[81] Wu-chang Feng, Ed Kaiser, and Travis Schluessler. Stealth measurements for cheat detection in on-line games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 15–20, 2008.

[82] Xiushan Feng and Alan J Hu. Cutpoints for formal equivalence verification of embedded software. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 307–316. ACM, 2005.

[83] PUB FIPS. 197: Advanced encryption standard (aes). *National Institute of Standards and Technology*, 2001.

[84] Robert W Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[85] Christophe Foket, Bjorn De Sutter, Bart Coppens, and Koen De Bosschere. A novel obfuscation: class hierarchy flattening. In *Foundations and Practice of Security*, pages 194–210. Springer, 2013.

[86] Stephanie Forrest, Anil Somayaji, and David H Ackley. Building diverse computer systems. In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pages 67–72. IEEE, 1997.

[87] Michael Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 workshop on New security paradigms*, NSPW '10, pages 7–16, New York, NY, USA, 2010. ACM.

[88] Vijay Ganesh, Sebastian Banescu, and Martín Ochoa. Short paper: The meaning of attack-resistant systems. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, pages 49–55. ACM, 2015.

[89] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, pages 519–531. Springer, 2007.

[90] S. Garg, C. Gentry, S. Halevi, M. Raykova, A Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Proc. of the 54th Annual Symp. on Foundations of Computer Science*, pages 40–49, 2013.

[91] Lynn E Garner. On the Collatz 3n + 1 algorithm. *Proceedings of the American Mathematical Society*, 82(1):19–22, 1981.

[92] Ian P Gent, Ewan MacIntyre, Patrick Prosser, Toby Walsh, et al. The constrainedness of search. In *AAAI/IAAI, Vol. 1*, pages 246–252, 1996.

[93] GitHub. Obfusc8: Implementation of Candidate Indistinguishability Obfuscation. `https://github.com/tum-i22/indistinguishability-obfuscation`. Accessed: 2017-03-03.

[94] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM. 01494.

[95] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[96] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[97] S. Goldwasser and G. N. Rothblum. On best-possible obfuscation. In *Theory of Cryptography*, pages 194–213. Springer, 2007.

[98] Pablo M Granitto, Cesare Furlanello, Franco Biasioli, and Flavia Gasperi. Recursive feature elimination with random forest for ptr-ms analysis of agroindustrial products. *Chemometrics and Intelligent Laboratory Systems*, 83(2):83–90, 2006.

[99] GuardSquare. ProGuard: The open source optimizer for Java bytecode. `https://www.guardsquare.com/en/proguard`. Accessed: 2017-03-03.

[100] Yoann Guillot and Alexandre Gazet. Automatic binary deobfuscation. *Journal in computer virology*, 6(3):261–276, 2010.

[101] Mark A Hall. *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.

[102] Maurice Howard Halstead. *Elements of software science*, volume 7. Elsevier New York, 1977.

[103] Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *SIGPLAN Not.*, 16(3):63–74, March 1981.

[104] Kelly Heffner and Christian Collberg. The obfuscation executive. In *International Conference on Information Security*, pages 428–440. Springer, 2004.

[105] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, 1981.

[106] Tony Hoare. The verifying compiler: A grand challenge for computing research. In *Joint Modular Languages Conference*, pages 25–35. Springer, 2003.

[107] Greg Hoglund. Hacking world of warcraft: An exercise in advanced rootkit design. *Black Hat*, 2006.

[108] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[109] Dominik Holling, Sebastian Banescu, Marco Probst, Ana Petrovska, and Alexander Pretschner. Nequivack: Assessing mutation score confidence. In *Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference on*, pages 152–161. IEEE, 2016.

[110] Bill Horne, Lesley Matheson, Casey Sheehan, and Robert E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Security and privacy in digital rights management*, pages 141–159. Springer, 2002.

[111] Susan Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):1–6, 1997.

[112] Amjad Ibrahim and Sebastian Banescu. Stins4cs: A state inspection tool for c. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, pages 61–71. ACM, 2016.

[113] Intel. Intel Active Management Technology — Query, Restore, Upgrade, and Protect Devices Remotely, 2016. `http://www.intel.com/content/www/us/en/architecture-and-technology/intel-active-management-technology.html`, [Online; accessed 20-September-2016].

[114] InterTrust. whiteCryption: Application Protection for a Hostile World. `https://www.intertrust.com/products/application-security/`. Accessed: 2017-03-03.

[115] Irdeto. Datasheet: Cloakware for Automotive - Software Protection Service. `https://irdeto.uberflip.com/solution-overviews-datasheets/datasheet-cloakware-for-automotive-software-protection-service`. Accessed: 2017-03-03.

[116] Matthias Jacob, Mariusz H Jakubowski, Prasad Naldurg, Chit Wei Nick Saw, and Ramarathnam Venkatesan. The superdiversifier: Peephole individualization for software protection. In *Advances in Information and Computer Security*, pages 100–120. Springer, 2008.

[117] Matthias Jacob, Mariusz H Jakubowski, and Ramarathnam Venkatesan. Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings. In *Proceedings of the 9th workshop on Multimedia & security*, pages 129–140. ACM, 2007.

[118] Jawahar Jain, Amit Narayan, Masahiro Fujita, and A Sangiovanni-Vincentelli. A survey of techniques for formal verification of combinational circuits. In *Computer Design: VLSI in Computers and Processors, 1997. ICCD'97. Proceedings., 1997 IEEE International Conference on*, pages 445–454. IEEE, 1997.

[119] Markus Jakobsson and Karl-Anders Johansson. Retroactive detection of malware with applications to mobile platforms. In *Proceedings of the 5th USENIX Conference on Hot Topics in Security*, HotSec'10, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association.

[120] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[121] Edward Kaiser, Wu-chang Feng, and Travis Schluessler. Fides: Remote anomaly-based cheat detection using client emulation. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 269–279, New York, NY, USA, 2009. ACM.

[122] Yuichiro Kanzaki, Akito Monden, and Christian Collberg. Code artificiality: a metric for the code stealth based on an n-gram model. In *Proc. of the 1st International Workshop on Software Protection*, pages 31–37. IEEE Press, 2015.

[123] 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, 2003.

[124] Matthew Karnick, Jeffrey MacBride, Sean McGinnis, Ying Tang, and Ravi Ramachandran. A qualitative analysis of java obfuscation. In *proceedings of 10th IASTED international conference on software engineering and applications, Dallas TX, USA*, 2006.

[125] Mohamed Karroumi. Protecting white-box AES with dual ciphers. In Kyung-Hyune Rhee and DaeHun Nyang, editors, *Information Security and Cryptology - ICISC 2010*, number 6829 in Lecture Notes in Computer Science, pages 278–291. Springer Berlin Heidelberg, January 2011.

[126] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.

[127] Auguste Kerckhoffs. *La cryptographie militaire, ou, Des chiffres usités en temps de guerre: avec un nouveau procédé de déchiffrement applicable aux systèmes à double clef*. Librairie militaire de L. Baudoin, 1883.

[128] J. Kilian. Founding crytpography on oblivious transfer. In *Proc. of the 20th Annual ACM Symp. on Theory of Computing*, pages 20–31. ACM, 1988.

[129] J. Kinder. Towards static analysis of virtualization-obfuscated binaries. In *19th Working Conference on Reverse Engineering (WCRE)*, pages 61–70, Oct 2012.

[130] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[131] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. Technical report, DTIC Document, 1951.

[132] Underwriters Laboratories. Ul 687 standard for burglary-resistant safes. `https://standardscatalog.ul.com/standards/en/standard_687`, 2011.

[133] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[134] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291. IEEE, 2014.

[135] Long Le. Payload already inside: datafire-use for rop exploits. *Black Hat USA*, 2010.

[136] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[137] Chu-Min Li and Bing Ye. Sat-encoding of step-reduced md5. *SAT COMPETITION 2014*, pages 94–95.

[138] Shih-Wei Lin, Zne-Jung Lee, Shih-Chieh Chen, and Tsung-Yuan Tseng. Parameter determination of support vector machine and feature selection using simulated annealing approach. *Applied soft computing*, 8(4):1505–1512, 2008.

[139] 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.

[140] Matias Madou, Bertrand Anckaert, Bruno De Bus, Koen De Bosschere, Jan Cappaert, and Bart Preneel. On the effectiveness of source code transformations for binary obfuscation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP06)*, pages 527–533. CSREA Press, 2006.

[141] 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, 2006.

[142] A. Main and Paul C. van Oorschot. Software protection and application security: Understanding the battleground. *International Course on State of the Art and Evolution of Computer Security and Industrial Cryptography,*, 2003.

[143] Anirban Majumdar. *Design and evaluation of software obfuscations*. PhD thesis, The University of Auckland New Zealand, 2008.

[144] Anirban Majumdar and Clark Thomborson. Manufacturing opaque predicates in distributed systems for code obfuscation. In *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*, pages 187–196. Australian Computer Society, Inc., 2006.

[145] Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Conqueror: tamper-proof code execution on legacy systems. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–40. Springer, 2010.

[146] Joshua Mason, Sam Small, Fabian Monrose, and Greg MacManus. English shellcode. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 524–533. ACM, 2009.

[147] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, November 2011.

[148] McAfee. McAfee Labs Threats Report. Technical Report March, 2016. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-mar-2016.pdf`.

[149] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[150] James P McDermott. Attack net penetration testing. In *Proceedings of the 2000 workshop on New security paradigms*, pages 15–21. ACM, 2001.

[151] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[152] Miles A McQueen, Wayne F Boyer, Mark A Flynn, and George A Beitel. Time-to-compromise model for cyber risk reduction estimation. In *Quality of Protection*, pages 49–64. Springer, 2006.

[153] Florian Merz, Stephan Falke, and Carsten Sinz. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In *International Conference on Verified Software: Tools, Theories, Experiments*, pages 146–161. Springer, 2012.

[154] Ilya Mironov and Lintao Zhang. Applications of sat solvers to cryptanalysis of hash functions. In *Theory and Applications of Satisfiability Testing-SAT 2006*, pages 102–115. Springer, 2006.

[155] Rabih Mohsen and Alexandre Miranda Pinto. Evaluating obfuscation security: A quantitative approach. In *International Symposium on Foundations and Practice of Security*, pages 174–192. Springer, 2015.

[156] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[157] John C. Munson and Taghi M. Kohshgoftaar. Measurement of data structure complexity. *Journal of Systems and Software*, 20(3):217–225, March 1993.

[158] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[159] Gleb Naumovich and Nasir Memon. Preventing piracy, reverse engineering, and tampering. *Computer*, (7):64–71, 2003.

[160] John A Nelder and R Jacob Baker. Generalized linear models. *Encyclopedia of statistical sciences*, 1972.

[161] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. Impact of community structure on sat solver performance. In *Theory and Applications of Satisfiability Testing–SAT 2014*, pages 252–268. Springer, 2014.

[162] Zack Newsham, William Lindsay, Vijay Ganesh, Jia Hui Liang, Sebastian Fischmeister, and Krzysztof Czarnecki. Satgraf: Visualizing the evolution of sat formula structure in solvers. In *Theory and Applications of Satisfiability Testing–SAT 2015*, pages 62–70. Springer, 2015.

[163] Vu Nguyen. *Improved size and effort estimation models for software maintenance*. PhD thesis, University of Southern California, 2010.

[164] MFXJ Oberhumer, László Molnár, and John F Reiser. Upx: the ultimate packer for executables, 2004.

[165] Enrique I Oviedo. Control flow, data flow and program complexity. In *Proc. IEEE COMPSAC*, pages 146–152, 1980.

[166] Jens Palsberg, Sowmya Krishnaswamy, Minseok Kwon, Di Ma, Qiuyun Shao, and Yi Zhang. Experience with software watermarking. In *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*, pages 308–316. IEEE, 2000.

[167] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. pages 601–615. IEEE, May 2012.

[168] Arash Partow. General Purpose Hash Function Algorithms. `http://www.partow.net/programming/hashfunctions`. Accessed: 2017-03-12.

[169] Karl Pearson. Note on regression and inheritance in the case of two parents. *Proc. of the Royal Society of London*, 58:240–242, 1895.

[170] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[171] John Platt et al. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

[172] PreEmptiveSolutions. DashO: Java & Android Obfuscator & Runtime Protection. `https://www.preemptive.com/products/dasho`. Accessed: 2017-03-03.

[173] PreEmptiveSolutions. Dotfuscator: .NET App Self Protection and Obfuscation. `https://www.preemptive.com/products/dotfuscator`. Accessed: 2017-03-03.

[174] Riccardo Pucella and Fred B Schneider. Independence from obfuscation: A semantic framework for diversity. *Journal of Computer Security*, 18(5):701–749, 2010.

[175] Jing Qiu, Babak Yadegari, Brian Johannesmeyer, Saumya Debray, and Xiaohong Su. Identifying and understanding self-checksumming defenses in software. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 207–218. ACM, 2015.

[176] Ganesan Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[177] Rolf Rolles. Program Synthesis in Reverse Engineering. `http://www.nosuchcon.org/talks/2014/D1_01_Rolf_Rolles_Program_Synthesis_in_reverse_Engineering.pdf`, 2014. NoSuchCon 2014, Accessed:2016-05-24.

[178] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[179] Aleieldin Salem and Sebastian Banescu. Metadata recovery from obfuscated programs using machine learning. In *Proceedings of the 6th Software Security, Protection and Reverse Engineering Workshop*, page 8. ACM, 2016.

[180] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[181] Florent Saudel and Jonathan Salwan. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, pages 31–54. SSTIC, 2015.

[182] T. Schneider. Practical secure function evaluation. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2008.

[183] Sebastian Schrittwieser and Stefan Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *Information Hiding*, pages 270–284, 2011.

[184] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)*, 49(1):4, 2016.

[185] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331. IEEE, 2010.

[186] SemanticDesigns. Thicket Family of Source Code Obfuscators. `http://www.semanticdesigns.com/Products/Obfuscators/`. Accessed: 2017-03-03.

[187] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 1–16. ACM, 2005.

[188] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[189] Adi Shamir and Nicko Van Someren. Playing 'hide and seek' with stored keys. In *Financial cryptography*, pages 118–124, 1999.

[190] M. Sharif, A Lanzi, J. Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 94–109, May 2009.

[191] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[192] KC Shashidhar, Maurice Bruynooghe, Francky Catthoor, and Gerda Janssens. Verification of source code transformations by program equivalence checking. In *International Conference on Compiler Construction*, pages 221–236. Springer, 2005.

[193] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.

[194] João P Marques Silva and Karem A Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227. IEEE Computer Society, 1997.

[195] Igor Skochinsky. IDA F.L.I.R.T. Technology: In-Depth. `https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml`, 2013. Accessed: 2017-03-03.

[196] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Information systems security*, pages 1–25. Springer, 2008.

[197] Frank Stevenson. Cryptanalysis of contents scrambling system. *Online publication http://www. dvd-copy. com/news/cryptanalysis-of-contents-scramblingsystem. htm*, 1999.

[198] Stunnix. C/C++ Obfuscator. `http://stunnix.com/prod/cxxo/`. Accessed: 2017-03-03.

[199] Iain Sutherland, George E. Kalb, Andrew Blyth, and Gaius Mulley. An empirical examination of the reverse engineering process for binary files. *Computers & Security*, 25(3):221–228, 2006.

[200] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[201] Symantec Corporation. Internet Security Threat Report 2016. Technical Report April, 2016. `https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf`.

[202] Kuo-Chung Tai. A program complexity metric based on data flow information in control graphs. In *Proc. of the 7th international conference on Software engineering*, pages 239–248. IEEE Press, 1984.

[203] PaX Team. Pax non-executable pages design & implementation. *Avaliable: http://pax. grsecurity. net*, 2003.

[204] Oreans Technologies. Code Virtualizer: Total Obfuscation Against Reverse Engineering. `http://oreans.com/codevirtualizer.php`. Accessed: 2017-03-12.

[205] Oreans Technologies. Themida: Advanced Windows Software Protection System. `http://oreans.com/themida.php`. Accessed: 2017-03-12.

[206] StrongBit Technologies. ExeCryptor: Stop crackers and software pirates. `http://www.strongbit.com/execryptor.asp`. Accessed: 2017-03-12.

[207] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon McCoy, Antonio Nappa, Vern Paxson, Paul Pearce, et al. Ad injection at scale: Assessing deceptive advertisement modifications. In *2015 IEEE Symposium on Security and Privacy*, pages 151–167. IEEE, 2015.

[208] Kurt Thomas, Juan Antonio Elices Crespo, Ryan Rasti, Jean-Michel Picod, Cait Phillips, Chris Sharp, Fabio Tirelo, Ali Tofigh, Marc-Antoine Courteau, Lucas Ballard, et al. Investigating Commercial Pay-Per-Install and the Distribution of Unwanted Software. In *USENIX Security Symposium*, 2016.

[209] S.K. Udupa, S.K. Debray, and M. Madou. Deobfuscation: reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering*, 2005.

[210] L. G. Valiant. Universal circuits (preliminary report). In *Proc. of the 8th Annual ACM Symp. on Theory of Computing*, pages 196–203. ACM, 1976.

[211] Valve. Valve Anti-Cheat System (VAC), 2015. `https://support.steampowered.com/kb_article.php?p_faqid=370`, [Online; accessed 20-September-2016].

[212] Mayank Varia. *Studies in program obfuscation*. PhD thesis, School of Computer Science, Tel Aviv University, 2010.

[213] Corrado Aaron Visaggio, Giuseppe Antonio Pagin, and Gerardo Canfora. An empirical study of metric-based methods to detect obfuscated code. *International Journal of Security & Its Applications*, 7(2), 2013.

[214] VMPSoft. VMProtect. `http://vmpsoft.com/products/vmprotect/`. Accessed: 2017-03-12.

[215] Z Vrba. cryptexec: Next-generation runtime binary encryption using on-demand function extraction. 2003.

[216] Chenxi Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *International Conference on Dependable Systems and Networks, 2001. DSN 2001*, pages 193–202, 2001.

[217] Huaijun Wang, Dingyi Fang, Ni Wang, Zhanyong Tang, Feng Chen, and Yuanxiang Gu. Method to evaluate software protection based on attack modeling. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 837–844. IEEE, 2013.

[218] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Security and privacy (SP), 2010 IEEE symposium on*, pages 497–512. IEEE, 2010.

[219] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear obfuscation to combat symbolic execution. In *Computer Security–ESORICS 2011*, pages 210–226. Springer, 2011.

[220] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[221] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Ph.D. thesis, Harvard University.*

[222] L. M. Wills. Automated program recognition: a feasibility demonstration. *Artif. Intell.*, 45(1-2):113–171, September 1990.

[223] Martin R. Woodward, Michael A. Hennell, and David Hedley. A measure of control flow complexity in program text. *IEEE Transactions on Software Engineering*, (1):45–50, 1979.

[224] Yongdong Wu, Hui Fang, Shuhong Wang, and Zhifeng Qi. A framework for measuring the security of obfuscated software. In *Proc. of 2010 International Conference on Test and Measurement*, 2010.

[225] Brecht Wyseur. *White-Box Cryptography*. PhD thesis, KATHOLIEKE UNIVERSITEIT LEUVEN, Kasteelpark Arenberg 10, 3001 Leuven-Heverlee, 2009.

[226] Yaying Xiao and Xuejia Lai. A secure implementation of white-box AES. In *2nd International Conference on Computer Science and its Applications, 2009. CSA '09*, pages 1–6, 2009.

[227] Babak Yadegari and Saumya Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, ser. CCS*, volume 15, pages 732–744, 2015.

[228] 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.

[229] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.

[230] yWorks. yGuard Java Bytecode Obfuscator and Shrinker. `https://www.yworks.com/products/yguard`. Accessed: 2017-03-03.

[231] Michal Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`.

[232] Yongxin Zhou, Alec Main, Yuan X Gu, and Harold Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In *International Workshop on Information Security Applications*, pages 61–75. Springer, 2007.

[233] Yan Zhuang and Felix Freiling. Approximating Optimal Software Obfuscation for Android Applications. In *Proc. 2nd Workshop on Security in Highly Connected IT Systems*, pages 46–50, 2015.

Glossary

AES Advanced Encryption Standard. 35

BDD Binary Decision Diagram. 19, 89, 90

BP Branching Program. 19–22, 24, 25, 183

CFF Control Flow Flattening. 42

CFG Control Flow Graph. 42, 55, 82, 83, 153

DRM Digital Rights Management. 4, 48

GCC GNU C Compiler. 28

IDA Interactive Disassembler. 66

IO input-output. 6, 28, 39

ISA Instruction Set Architecture. 38, 39, 41, 155

LLVM Low Level Virtual Machine. 28

LOC Lines of Code. 93, 101, 103, 111, 112, 114, 126, 127, 147

LUT Look-up Table. 35

MATE Man-At-The-End. xi, 3–7, 9–11, 13, 15, 17, 18, 27, 30, 31, 36, 40, 41, 43, 44, 47–56, 60, 62, 64–66, 69, 71, 72, 76, 79–84, 89, 107, 108, 129, 130, 132, 145–150, 152–157, 183, 184, 187

MBA Mixed Boolean-Arithmetic. 34, 35, 38

MITM Man-In-The-Middle. 3, 50, 63

MJP Multi-linear Jigsaw Puzzle. 21, 22

PPT Probabilistic Polynomial Time Turing Machine. 16–18

PUP Potentially Unwanted Program. 7, 64

RBP Randomized Branching Program. 20–22, 24, 25, 183

ROP Return Oriented Programming. 37, 38, 42

SAT Boolean satisfiability. 85, 87–91, 94, 100, 101, 103, 105, 113–120, 125–128, 134, 155, 156, 184, 188

SHA Secure Hash Algorithm. 36

SMT Satisfiability Modulo Theories. 84, 85, 87–92, 94, 96–101, 103–105, 113, 114, 125, 127, 128, 132, 133, 156

SVM Support Vector Machine. 69, 70, 72, 109

UBP Universal Branching Program. 21, 22

UC Universal Circuit. 20–22, 24, 25

UCC Unified Code Counter. 93, 96, 101, 103, 111, 114, 116, 117, 120, 122–124

WB-AES White-Box Advanced Encryption Standard. 35

WBC White-Box Cryptography. 35, 74

WB-DES White-Box Data Encryption Standard. 35

Index

List of Figures

Listings

List of Tables