Classification and Static Detection of Obfuscated Web Application Backdoors

A thesis submitted to the
Graduate School of Natural and Applied Sciences

by

Furkan Paligu

in partial fulfillment for the
degree of Master of Science

in

Cyber Security Engineering

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science in Cyber Security Engineering.

APPROVED BY:

Prof. Dr. Ensar Gül ....
(Thesis Advisor)

Dr. Ferhat Özgür Çatak ....
(Thesis Co-advisor)

Yrd. Doç. Dr. Ahmet Bulut ....

Yrd. Doç. Dr. Betül Demiröz ....

This is to confirm that this thesis complies with all the standards set by the Graduate School of Natural and Applied Sciences of İstanbul Şehir University:

DATE OF APPROVAL: 08.01.2018

SEAL/SIGNATURE:

Declaration of Authorship

I, Furkan PALIGU, declare that this thesis titled, 'Classification and Static Detection of Obfuscated Web Application Backdoors' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date: 14/01/2018

" Supreme excellence consists in breaking the enemy's resistance without fighting. "

Sun Tzu

Classification and Static Detection of Obfuscated Web Application Backdoors

Furkan Paligu

Abstract

Backdoors, which are methods of bypassing authentication, are great enemies of secure software development process, especially when the risk factors indicate high requirements for security precautions. Unfortunately, the detection techniques of backdoors in web application software are imperfect and the attacker motivation is high. Furthermore, the obfuscation techniques make the detection much more difficult and time consuming than the traditional cases. In this study, we are proposing a classification of obfuscation techniques on web application backdoors based on 200 backdoor instances and the reported cases of backdoor detection on web applications. The study also includes the detection rates of static code vulnerability analyzers on backdoors and the obfuscation techniques applied to them. A conclusion is drawn on the effects of the classified obfuscation techniques to the static detection and how to improve the detection rates in the cases of obfuscation.

Keywords: Security, Static Code Analysis, Backdoors

Kamufle Edilmiş Web Uygulama Arka Kapılarının Sınıflandırılması ve Statik Tespiti

Furkan Paligu

Öz

Kimlik doğrulamayı atlatma teknikleri olan arka kapılar, özellikle risk faktörleri yüksek güvenlik önlemleri gereksinimlerine işaret ettiğinde, güvenli yazılım geliştirme sürecinin büyük bir düşmanıdır. Ne yazık ki, web uygulama yazılımlarında arka kapı tespit teknikleri yetersiz ve saldırgan motivasyonu oldukça yüksektir. Ayrıca, kamuflaj yöntemleri tespit sürecini olduğundan daha zor ve zaman alıcı bir duruma getirmektedir. Bu çalışmada, 200 arka kapı örneğinin incelenmesi ve daha önceden raporlanmış olan arka kapı tespiti olaylarının incelenmesi odağında bir arka kapı kamuflaj yöntemleri sınıflandırması öneriyoruz. Çalışma ayrıca popüler statik kod açıklık analiz araçlarının arka kapılar ve kamuflaj yöntemleri üzerindeki performans incelemelerini içermekte ve tespit oranlarının arttırılması için etkili bir prosedür önermektedir.

Anahtar Sözcükler: Güvenlik, Statik Kod Analizi, Arka Kapılar

I dedicate this work to Merve Astekin who has been a great friend and a wonderful confidant during my work

Acknowledgments

I would first like to thank my thesis co-advisor Dr.Ferhat Özgür Çatak who spent a great deal of time for me during this work and kindly showed me the way whenever I needed it. I also would like to thank my advisor Prof. Dr. Ensar Gül for his valuable contributions to this work.

Contents

Declaration of Authorship ii

Abstract iv

Öz v

Acknowledgments vii

List of Figures x

List of Tables xi

Abbreviations xii

1 Introduction 1
- 1.1 Backdoors . . . 1
- 1.2 Motivation for Backdoor Studies . . . 1
- 1.3 Types of Backdoors . . . 2
  - 1.3.1 Backdoors in a cryptographic algorithm . . . 3
  - 1.3.2 System Level Backdoors . . . 3
  - 1.3.3 Application Backdoors . . . 3
- 1.4 A Brief History of Backdoors . . . 4
- 1.5 Contribution . . . 5

2 Literature Review 6
- 2.1 Compiler Backdoors . . . 6
- 2.2 Application Backdoors . . . 7
- 2.3 Client Side Backdoors . . . 7
- 2.4 Binary Backdoors . . . 8
- 2.5 Backdoors at Malware Classification Systems . . . 9

3 Methodology 12
- 3.1 Collecting Backdoor Samples . . . 12
- 3.2 Experimental and Observational Environment . . . 13
- 3.3 Selection of Static Code Vulnerability Analysis Tools . . . 15
- 3.4 Running Analysis . . . 16

4 Classification of Backdoor Obfuscation Techniques 17
- 4.1 Run Time Parameter Modification . . . 18

4.2 Intentional Vulnerabilities Left on Software 21
4.3 Multistage Backdoor Insertion 22
4.4 Hidden Logic Flow and High Code Complexity 22
4.5 Hiding Malicious Content on Indirect Data 25
4.6 Client Side Backdoor Obfuscation 25
4.7 Post Analysis Backdoor Generation 27
4.8 Embedding Malicious Binary to Software 28

5 Evaluation of Static Code Vulnerability Analyzers on Backdoors 29
5.1 YASCA v2.2 34
5.2 RIPS v0.55 34
5.3 Visual Code Grabber v2.1.0.0 34
5.4 RATS v2.3 35

6 Conclusions and Future Work 36
6.1 Conclusions 36
6.2 Future Work 37

Bibliography 38

List of Figures

3.1 Database ER Diagram 14

4.1 Result of static code analysis run byRIPS on a simple backdoor using passthru function with direct parameters. 19
4.2 Result of static code analysis run by RIPS on a simple backdoor using passthru function with obfuscated parameters 21
4.3 McCabe's Code Complexity Analysis Graphs - 1 23
4.4 McCabe's Code Complexity Analysis Graphs - 2 24

5.1 Detection rates of obfuscated and not obfuscated backdoors 32
5.2 Comparison of the detection rates after the procedure 33

List of Tables

2.1 Classification of Backdoors in Wysopal 8
2.2 Kaspersky Classification of Malware Programs 10
2.3 Generic Malware Classification 11

3.1 PHP Tools for Static Code Analysis 15
3.2 Java Tools for Static Code Analysis 15
3.3 ASP Tools for Static Code Analysis 16

4.1 Backdoor Obfuscation Techniques Aggregated from Web 18
4.2 Backdoor Obfuscation Techniques Observed in This Study 19
4.3 Summary of Backdoor Obfuscation Classification 20

5.1 Static Detection Rates on PHP Backdoor Samples 31
5.2 Static Detection Rates on ASP and Java Backdoor Samples 31

Abbreviations

DES	**D**ata **E**ncryption **S**tandard
NSA	**N**ational **S**ecurity **A**gency
NBS	**N**ational **B**ureau of **S**tandards
IOS	**i**Phone **O**perating **S**ystem
XSS	Cross **S**ite **S**cripting
SQL	**S**tructured **Q**uery **L**anguage
ASCII	**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
ASP	**A**ctive **S**erver **P**ages
JSP	**J**ava **S**erver **P**ages
SMTP	**S**imple **M**ail **T**ransfer **P**rotocol
HTTP	**H**yper **T**ext **T**ransfer **P**rotocol
UDP	**U**ser **D**atagram **P**rotocol
ICMP	**I**nternet **C**ontrol **M**essage **P**rotocol
PDF	**P**ortable **D**ocument **F**ormat

Chapter 1

Introduction

1.1 Backdoors

Sun Tzu said; ”What enables the wise sovereign and the good general to strike and conquer, and achieve things beyond the reach of ordinary men, is foreknowledge. This foreknowledge cannot be elicited from spirits; it cannot be obtained inductively from experience, nor by any deductive calculation. Knowledge of the enemy's dispositions can only be obtained from other men, hence the use of spies” [1].
Backdoors which are malicious software that is used for bypassing security controls to access systems and the sensitive information on these systems are the substitute of usage of spies in the 5th century. They are widely used in almost every digital platform involving a wide range of actors from world's top companies to the strongest nations [2], serving as the spies of the digital world.

1.2 Motivation for Backdoor Studies

Usage of malware at its very beginning was about the protection of the copy rights of the software. But it evolved and turned into a wedge allowing its creator to gain recognition and make good profit [3, 4]. Backdoors as the other types of malware offer that gain to the developers and come with a very easy to implement nature, making it a dangerous concept.
It is a simple and straightforward process to insert a backdoor within an application.

Depending on the hiding technique, it takes none to very little effort. An attacker, who places a backdoor into the code, is able to obtain access to the system anytime without the difficult process of penetration. This access grants many functions such as remote control and espionage activities which in many cases goes unnoticed for years of operation. Hence the detection of backdoors is an exceptionally significant concept for the secure software development process.
Unfortunately, when backdoors exist within a legitimate web application, it is very difficult to detect them using dynamic analysis, regular network activity of the application often covers for the flow of information in between the server and the third parties. At the same time, the static detection techniques on software are imperfect. Vulnerabilities are commonly obfuscated by creative methods which weakens the effectiveness of automated analysis. Furthermore, human review is inconveniently time consuming and thus expensive. Therefore, there is a strong need for research and development on backdoor detection and prevention.

1.3 Types of Backdoors

Unwanted software that deliberately causes major security threats to the computer systems and their users is defined as malicious software or malware [5]. Backdoors are unwanted software that allows a person access to a computer system without user consent or knowledge, therefore by definition and structure, all types of backdoors are malware. Backdoors can emerge unintentionally as programming mistakes, or they can be planted in the software as an inside job which makes them deliberate backdoors. As it is stated in Linux Information Project, deliberate backdoors can be planted in two ways; with the consent of the company authorities, intentionally ordering the plantation, or by the employees without any knowledge at the managements part [6]. Therefore some backdoors which are deliberate are also legitimate. They are used in extraordinary circumstances such as obligatory information sharing, debugging the software, or providing control over passwords for particular cases. It is also common to use backdoors for testing and validation of the applications. In these cases it is very important to ensure that the version of the software being released is cleared of all related functionalities.
We have also encountered cases where backdoors are planted into the software or the system by an outside attacker who discovers a vulnerability in the system and exploits

it by injecting code or various objects into the system. In some cases files injected into the system searches for specific types of code files and then injects them with malicious code turning them into backdoors.
In literature there is also the distinction of symmetric and asymmetric backdoors [7]. This distinction is based on the ability to use the backdoor. A backdoor that anyone can use is a symmetric backdoor whereas, the backdoors that can only be used by the person that create it are called asymmetric backdoors.
Backdoors exist in three levels; application level backdoors, system level backdoors, and the backdoors that exist in cryptographic algorithms [8].

1.3.1 Backdoors in a cryptographic algorithm

In cryptographic algorithms backdoors are designed to undermine the underlying mathematics of cryptography as they incorporate a number of deliberately inserted errors and vulnerabilities designed to make cryptanalysis easier [9]. This type of backdoors have gained popularity in 1990s and 2000s as the modern encryption emerged with the growing usage of commercial internet

1.3.2 System Level Backdoors

Backdoors in system level provide an access to the operating system operations and data. A real life attempt at a system level backdoor was the Linux kernel incident where a function, when being called under particular cases, provided root privileges to the user [10]. Backdoors of operating systems such as Windows, Linux and IOS (iPhone Operating System) are system level backdoors

1.3.3 Application Backdoors

Finally, the application backdoors are planted access points in an application disregarding security mechanisms employed during the development process. Backdoors at the application servers, or the client side that run the application code is in this category. Application backdoors are very easy to create using various programming languages and tools, and difficult to detect considering the size and the prevention measures of the software. A simple application used in a small company is often not cheeked against

these threats as a result of the lack of awareness and limited resources. However when the applications are granted access in the system, the impact is very severe.

1.4 A Brief History of Backdoors

The first study on the malware is recorded in many sources [11, 12] as Von Neumann introducing the idea of Automata Theory in 1949 with. He explained the self replicating machine in a cellular automata and how small replicates of programs taking over others [13].

From that point, the academic work on backdoors are dating back to 1967 where the term trap door has been used. In the paper, Petersen and Turn discusses the active infiltrators utilizing the weak entry points in the system inserted by the corrupt developers [14]. In the paper, the distinction of accidental and deliberate backdoors is also clearly stated.

Rapid encounters of the crypto backdoors has primarily started with the rise of the internet in 1990s where crypto algorithms has been deployed for encryption of the information flow. However, about 15 years before that the creation of the DES (Data Encryption Standard) has been a point to discuss whether NSA (National Security Agency) has deliberately let the algorithm contain weaknesses as a deliberate backdoor. As a result, a workshop was organized by NBS (National Bureau of Standards) to analyze the backdoor concern [15, 16].

In 1993 the US government proposed Clipper Chip which included a deliberate backdoor and raised the early debates over personal safety and national security against privacy of personal information [17].

Application side of the backdoor history has shown growth with the rapid raise of the applications. In 2002 tcpdump utility of Linux has been attempted to be inserted with a backdoor, where it could be activated through port 1963 [18]. Fortunately, it has been found easily and rooted out form Linux systems. As backdoors in Joomla commercial and WordPress plug-ins were exposed, a door opened that can never be truly closed. Applications that we use everyday regularly, where we share our most intimate information are under risk of leaking, changing and manipulating information.

1.5 Contribution

There are three main contributions of this thesis to the literature. They are in the categories of structure and the state of detection.

The first output is based on backdoor structure, it is a list of obfuscation techniques that has been observed on the real life samples of web application backdoors gathered from the code search engines. These observed teqhniques are merged into Table 4.1 with techniques that are already published in literature in order to provide a comprehesive source of information. For each listed item, it is stated whether the information is reached by the examination of the samples or the literature check or both. In Chapter 4, each of these techniques, regardless of it's source, is explained in detail.

The second output is based on the current state of detection by static code analysis tools. The detection performances of the common open source static code vulnerability analyzers has been listed in Chapter 5, including a comparing analysis of the detection on obfuscated and plain samples.

Finally, each class of obfuscation has been heavily scrutinized and the detection rates of the tools have been improved by a suggested procedure on the backdoor instances. Leaving an essential structure for the further studies on backdoor obfuscation and the static detection techniques.

Chapter 2

Literature Review

Academic studies on the application backdoors are rare. Obfuscation methodologies or the static code analysis points of view on the web application backdoors is even more scarce. There are a number of studies that discuss compiler issues, JavaScript, PDF backdoors, and backdoors as computer viruses where the inspector does not have direct access to the source code. In this chapter we discuss some of the most related works to this study.
Beside academic studies, a number of software security and anti-malware companies have their own perspective publishing on various sites, which is a good supportive source on the subject especially considering the scarce nature of the studies. A classification point of view of backdoors from these sources are mentioned at the end of this chapter.

2.1 Compiler Backdoors

The most important work on the backdoors is Ken Thompson's 'Reflections on Trusting Trust' [19]. The paper is exposing the risk that comes with the compilers, and urging the need to not to trust the code that you did not totally create yourself. The main idea is that when a developer implement a logic into the program and compile it in a source that is not known, there is a risk that the compiled source will be altered in the compiler. Thus it may contain malicious content without the developer knowing about it. This study however is for the self modifying compiler code and different than application

backdoors in this study as we focus on the backdoor software and the compiler that handles it.

2.2 Application Backdoors

One of the most noteworthy studies on the subject is [8] where Wysopal, Eng and Shields has made a clear classification of application backdoors. Furthermore, states effective detection strategies for each one of them. Table 2.1 lists the classification of backdoors from Wysopal along with their short definitions and suggested detection strategies.
In Wysopal, the primary focus is to list the classes and methodologies with common obfuscations and suggested detection strategies. It does not however, give an inspection of large number of samples and methodological evaluations performed on them. In this sense, the study distinguishes from this one. Moreover, the primary focus of this study is on the obfuscation techniques of the backdoors.

2.3 Client Side Backdoors

As discussed in Chapter 4 in detail, the client side script are very convenient to hide backdoors. Therefore, it is easy to find studies to focuses on client and in particular JavaScript code. Moreover, it fairly easier to find malicious JavaScript code on web because anti-virus programs and cites gives access to the web site code that is scanned against viruses.
A number of studies have been performed to detect backdoors in JavaScript code using various techniques such as methods that help discover JavaScript automatically where obfuscation is applied. Commonly using metrics such as "N-gram", "Entropy", and "Word Size" [20].
There are also several methods for detecting malicious flow dynamically, although they are generic and specialized on certain attack types [21, 22]. However, higher level systems that examines the patterns in various methods can be found in [23–26].

Table 2.1: Classification of Backdoors in Wysopal

Class	Short Definition	Detection Strategies
Special credentials	Special credentials and logic intentionally inserted into the program code	Detecting variables that hold values similar to usernames and passwords
Hidden functionality	The attacker inserts a logic that helps bypass the authentication procedure and execute commands	Recognizing common patterns in scripting languages
Unintended network activity	Involves several techniques such as, espionage activities on ports that are not documented, establishing connections to external sources in order to give them command and control, and leaking sensetive data through variousprotocols	Identifying inbound and outbound connections. Looking for hard-coded IP address or ports and analyze the data flow Identify potential information leaks. Examining file system and registry I/O Profiling binaries by examining import tables.
Manipulation of security critical parameters	Using manipulations on the critical parameters, and deficient logic comparisons the attacker is able to change the program behaviour into a way he wants.	Creating a list of the critical variables and scrutinizing them with every reference. Examining known behavioral patterns

2.4 Binary Backdoors

Binary techniques of embedding backdoors are also effective self-contained obfuscation techniques in their nature. Unfortunately the number of studies on the backdoors and their obfuscation techniques is very limited. Even though, when the extend narrows to a specific area such as binary backdoors, the situation is much more brighter, there is still a big gap for the studies to carry on. Furthermore, the studies are mostly focused on a particular view such as defense and detection algorithms. The detection techniques of backdoors without obfuscation nor field complications is a common topic among the existing studies for binary specific backdoors and their defensive strategies.

In [27] the input files are being distinguished between malicious types of backdoor, virus, worm, trojan, constructor from benign files of types DOC, EXE, JPG, MP3, PDF and ZIP just on the basis of byte-level information. The results of the experiments in this

study shows that the technique achieves 90 % detection accuracy, which is promising against binary obfuscation when it is applied without further obfuscation.

In [28], In order to distinguish benign and malware executables in Windows and MS-DOS platforms, a variety of data mining methods are used. With one of the feature extraction approach overall detection accuracy of 97.11 % is achieved. Several other studies that uses data mining and N-gram techniques for malware detection and file type identification can be found at [29–32].

The creative methods of injecting the binary source, generated after compilation of the source code, is discussed in detail in Chapter 4.

2.5 Backdoors at Malware Classification Systems

Making a classification on malicious software is a difficult task. Every year a large number of malware is released and each year they are modified and evolved a bit more. Maintaining and updating the classification structure requires enormous effort. In a quarterly threat report of McAfee in 2014, it is stated that in one quarter more than 350 million of malicious code instances have been entered to the company servers and 50 million of these were new instances [33]. Microsoft also stated in 2014 that computers with its protection and detection software worldwide inspect over 700M computers monthly, which yields to the inspection of tens of millions of potential malicious software each day [34]. Therefore the classification made in the studies that are mentioned in this section may soon be updated and changed by the companies that have made them.

In [35] it is declared that the Kaspersky Lab was using a classification of malware based on the behavior, sub-behavior, operating system and the modification of the malicious programs. Modification stands for the different versions of a malicious program that are grouped under one name. OS stands for the operation system it works on. The name parameter here is given to the malicious program by the team. The reflection of the summary of the classification on the final name of the malicious program goes as Behaviour[-Sub-behaviour].OS.Name[-Modification:]. In this classification backdoors are listed as a behavior of the malware rather than a category itself.

In 2013 However, Kaspersky published a new malware classification system in which the backdoors are have their own class described as a remote administration tool that allows a person access to a computer system without user consent or knowledge [36].

The classification Kaspersky on malware programs be found at 2.2.

Table 2.2: Kaspersky Classification of Malware Programs

Class	Description
Viruses	Programs that are installed in existing programs without user consent and replicates(injects) themselves to other programs
Worm	Worms differ from viruses as they do not infect existing files but rather installed and exist as their own programs
Trojan	Programs that conceal themselves under another function but operate as a malware program of which can have the function of another type of malware program such as a backdoor or a downloader
Ransomware	Scam programs that try to take money from the users by showing up on a website and triggering a vulnerability on the client system and/or at a later stage, scaring the users into sending money
Rootkit	Programs that operates secretly and able to penetrate deep into the operating system using its functions
Backdoor	An application that allows a person access to a computer system without user consent or knowledge. It is able to launch other software, send/receive information, delete files and use the functions and the utilities of the infected system such as microphones and cameras.
Downloader	Infects through most commonly email attachments and malicious images and downloads additional, more complicated malware into the system

In a study of classification based on the obfuscation, it is mentioned that in the modern forms of the studies, a sample of malicious code can be in more than group, which is very common. Therefore, first a generic classification needs to be performed in which a malware can belong to more than one group [11]. In this classification backdoors are listed under the class of hidden malware. This generic classification can be found at Table 2.3.

In a different study also suggesting that a malicious code can belong to more than one of the groups, which is a survey of analysis and classification, variations of malware is put down as "Virus, Worm, Trojan-horse, Rootkit, Backdoor, Botnet, Spyware, Adware" [37].

It appears that from many work of classification raises a common ground for backdoors, rootkits, viruses, worms and trojans with differences for the rest of the classification. It is also mentioned in [38] that even though various classifications for malicious code exists, they always have a common ground as classes overlap and many times subclasses

are related.

TABLE 2.3: Generic Malware Classification

Malware Type	Malware Subtype
Propagation	• Virus • Worm
Lucrative	• Spyware • Ransomware • Scareware • Bot • Adware • Dialers
Hidden	• Backdoors • Trojans • Rootkits

Chapter 3

Methodology

The primary notion of the experiments in this study is to draw conclusions on the backdoors and their obfuscation techniques. From this base, the process has been initiated with the collection of backdoors so that the study input would have been obtained. Next, the experimental and observational environment for sample examination has been set up. Additionally, in order to determine the current state of detection and the ability of static code vulnerability analysis on backdoors, a set of tools has been examined and selected based on their relevance to the study.

After the basis for the research has been set, obfuscation techniques applied on the backdoor instances, and the ability of the static code vulnerability analyzers to detect these obfuscations has been examined.

3.1 Collecting Backdoor Samples

The process of collecting backdoor samples is not an easy task; publishing a sample or a collection of backdoors on the open web is commonly perceived to be immoral. The reason for that is simple, any malicious code that is shared on the web is likely to end up in an operational software application somewhere. Therefore, there is a lack of research databases for the researchers to perform experiments.

The best places to look when the instances cannot be openly published is the code search engines, where the samples are gathered as pieces of codes from different sources that is searched with proper key phrases and file extensions.

The backdoor instances examined in this study includes 159 PHP, 30 ASP (Active Server Pages) and 11 JSP (Java Server Pages) files, which has been gathered from GitHub and SearchCode. The main reason that the collection includes more PHP files is because PHP backdoor samples are more commonly shared in code search engines. Furthermore, they can be gathered relatively easier from code pieces of a file and run with much less dependencies.

3.2 Experimental and Observational Environment

Once the backdoor samples had been gathered, it was clear that there was a need for a platform on which the experiments and observations could be performed. The files of backdoors was suppose to be upgraded while the connection to the original is kept, so that any change (most likely an obfuscation applied) could be observed in terms of operation and static detection.

An application has been developed to keep track of the files and the results of the static code analysis performed on them. The application consists of a PHP program, a MySQL database and a set of desktop files. Backdoor instances is kept in the file system while information about them including their location, connected versions and static code analysis results is recorded to the database.

Program displays the selected files on screen, saves the changes with connected version, making a record both on the database and the file system. Thus, the results of the analysis can be saved where it is easy to draw conclusions with the connected files.

An automated static code analysis process has not been realized during this study because of the straightforward process that does not need automation for the initial studies. It is however a beneficial approach that can save time for the future studies on the database created. Instead, in this study the analysis has been triggered manually and results has been produced as html and txt files. However, the process of transferring the results from the output files to the database is realized by the program automatically.

ER diagram of the database designed for this study can be found in Figure 3.1. It includes tables for obfuscations techniques detected in the samples with the connection to them on the records table.

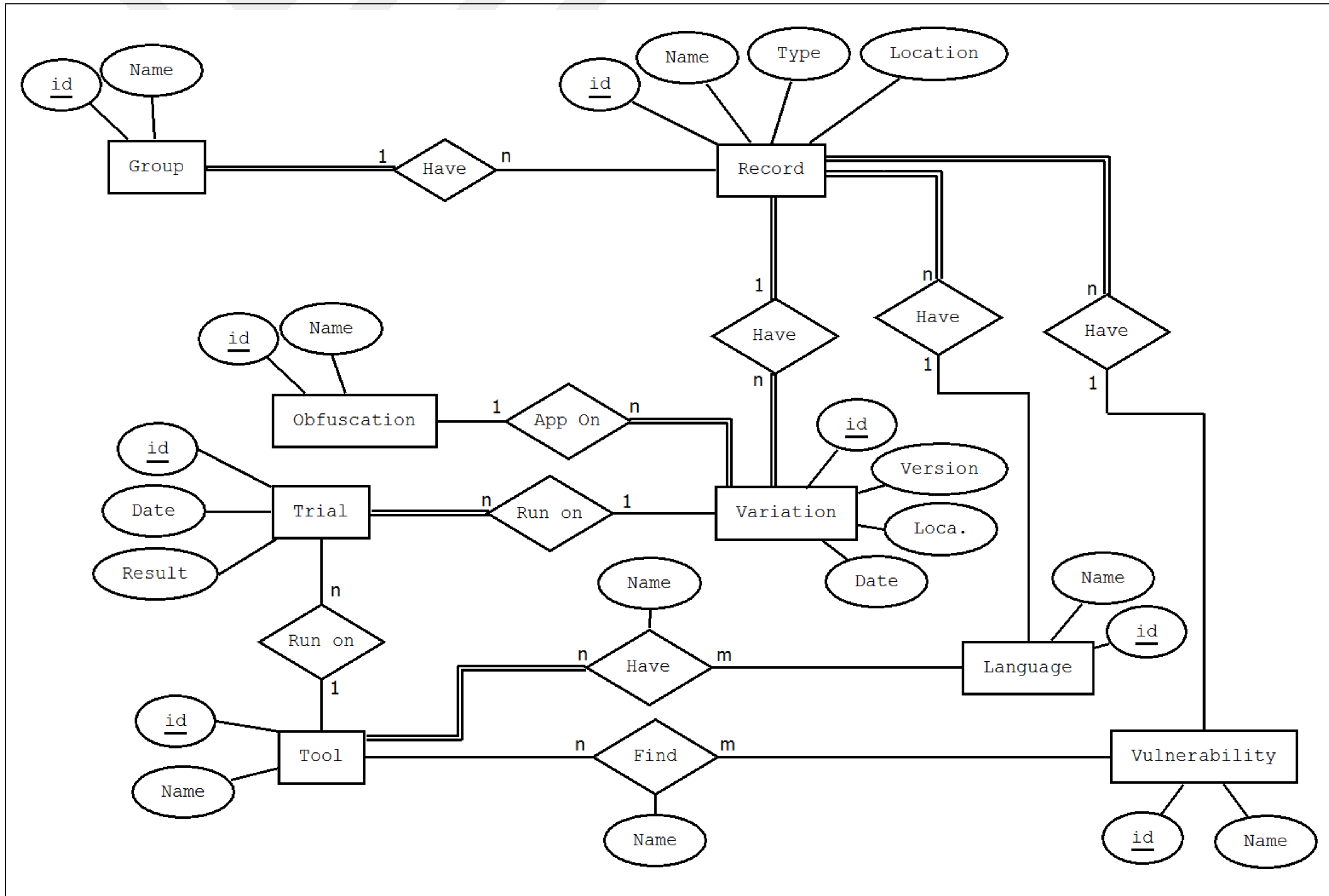

FIGURE 3.1: Database ER Diagram

3.3 Selection of Static Code Vulnerability Analysis Tools

Selection of the open source static code vulnerability analyzers has been carried out based on the backdoor samples collected and their programming languages. The listed source code security analyzers from [39–41] has been experimented on a few samples for starters to eliminate the irrelevant tools. It appeared that a vast majority of the open source static code analyzers are not suited for backdoor detection nor the detection of related vulnerabilities. However, after careful deliberation we decided on 4, which gives best backdoor oriented results, to include in this review. All the analyzers in this section operate on PHP, and only Visual Code Grabber can operate on ASP and JSP files.
Table 3.1, 3.2 and 3.3 lists the results of an elimination process for the open source code security analyzers that is examined. Tables do not include the results for all tools from the elimination as only relevant programming languages and vulnerability concepts has been further examined.

Table 3.1: PHP Tools for Static Code Analysis

Name	Outcome	Notes
RIPS	Suitable for this study	
YASCA	Suitable for this study	
RATS	Suitable for this study	
VisualCodeGrepper	Suitable for this study	
PHP-sat	Inadequate	Only a few vulnerabilities related to backdoors
PIXY	Irrelevant	Only for XSS, SQL Injection, File Inclusion vulnerabilities
DevBug	No additional contribution	Uses RIPS in the background which is already in the study
WAP	Irrelevant	Input validation related

Table 3.2: Java Tools for Static Code Analysis

Name	Outcome	Notes
VisualCodeGrepper	Suitable for this study	
OWASP ORIZON	Inadequate	Very new
OWASP WAP	Irrelevant	Only Input validation
OWASP Code Crawler	Inadequate	Very new, no documentation
PMD	Irrelevant	Detects dead code, duplicate code
FindBugs	Irrelevant	Detects possible bugs
LAPSE	Inadequate	Only a few vulnerabilities related to backdoors

Table 3.3: ASP Tools for Static Code Analysis

Name	Outcome	Notes
VisualCodeGrepper	Suitable for this study	
YASCA	Inadequate	Needs plugins
.NET Security Guard	Inadequate	Only a few vulnerabilities related to backdoors

3.4 Running Analysis

First, for the listing of observed obfuscation techniques, every backdoor sample has been opened and scrutinized to understand the method of the backdoor and the method of the obfuscation if any applied. The results has been noted and grouped under the Table 4.2 in Chapter 4.

The analysis run by using the static code vulnerability analysis tools has been far more problematic. First of all, the result set from every single tool was different. That meaning not only the format of the output file, but the content in which the vulnerabilities are referred. Second, since the backdoor files are not designed to be secure software, security measures for the coding beside the original function of the intended backdoor has reflected on the results. For instance, if a command line executioner is using a command it gets with the GET method, security issues with the usage of the GET method is all documented in the result outputs. Considering that in many of the backdoor samples there are over hundreds of lines of code to be used after the exploitation of the system, it is easy to see that the results got very messy.

Another issue was the reflection of the vulnerabilities caused by the backdoors to the reports, since a lot of the critical functions, used in the backdoors, are also used in the legitimate coding practices, tools avoided calling what they catch backdoors or vulnerabilities but rather warnings of possible weak points. Therefore, in order to proceed in the studies, any warning or notice given on the point of the backdoor vulnerabilities has been accepted as a successful detection of the backdoor. This is also the reason why false positive analysis are not included in the study.

Chapter 4

Classification of Backdoor Obfuscation Techniques

This section lists common backdoor obfuscation classes based on the techniques obtained from the examination of instances and reported cases of detection. These techniques are commonly applied in software and can be combined and cultivated into further sophisticated methodologies.

Backdoors on the compilers that change the code on compilation cannot be classified and detected making analysis solely on the software. The subject of this classification is on the application code developed, and not the compiler that handles it. Therefore, malicious compiler issues are not covered in this study.

The studies and reported cases of backdoor obfuscation cases that are published in web are aggregated in Table 4.1 which includes inspection of hacked websites and scrutinization of supicious behaviour. In Table 4.2 we have listed the backdoor obfuscation techniques that is observed in this study (one by one inspection of 200 samples of real life backdoors). These tables are put together in Table 4.3 to give the summary of the classification proposed in this thesis. It includes how each class is encountered, and the methodologies of obfuscation classes that are observed in the samples examined.

The symbol 'X' is used in the 'Observed Methodologies in Backdoor Instances' column in order to indicate the cases where the classification is based on reported cases on open web, without any encounter on our samples.

TABLE 4.1: Backdoor Obfuscation Techniques Aggregated from Web

Class of Obfuscation	Encountered Methodologies
Run Time Parameter Modification	• Encrypting functions • Sophisticated string operations such as keyword substitution, string splitting, character picking
Intentional Vulnerabilities	• Deficiencies left on input filtering mechanisms • Intentional mistakes on logic operations • SQL and command injection • File inclusion
Multistage Backdoor Insertion	• Software updates that transform the code into a backdoor
Hidden Logic Flow and High Code Complexity	• Alterations and additions on conditional branches • Creating unreadable methods
Hiding Malicious Content on Indirect Data	• Collecting critical data from external files • Running third party executables inserted in unusual files
Client Side Backdoor Obfuscation	• String manipulations of JavaScript code • Sending malicious code with PDF executables
Embedding Malicious Binary to Software	• Injecting random binaries by using code caves in executables • Binary protecting methods of packer tools

4.1 Run Time Parameter Modification

Critical functions that are most commonly used in backdoors often must be used in the software for legitimate purposes as well. In particular, web applications use critical functions to trigger external programs in order to provide services and access points for exhausting operations. In this obfuscation technique, the attacker intention is to hide the malicious execution of the critical functions by obfuscating the parameters of operation, making them difficult to be detected during automated static analysis and human review. In the experiments we run using PHP code pieces, backdoors using passthru function with direct parameters has been easily detected by several tools. However, with a little effort of hiding the parameters, the detection was easily bypassed. In Figure 4.1 and 4.2, there are results of static analysis on two different usage of passthru function by RIPS analysis tool. In Figure 4.1, the parameter of the function is plain text and in Figure 4.2 it is obfuscated by a simple Base64 encryption.

An inside attacker may also attempt to obfuscate embedded shell commands using the

TABLE 4.2: Backdoor Obfuscation Techniques Observed in This Study

Class of Obfuscation	Observed Methodologies in Backdoor Instances
Run Time Parameter Modification	• Encrypting functions • Sophisticated string operations such as keyword substitution, string splitting, character picking
Intentional Vulnerabilities	• Deficiencies left on input filtering mechanisms • SQL and command injection • File inclusion
Hidden Logic Flow and High Code Complexity	• Alterations and additions on conditional branches
Hiding Malicious Content on Indirect Data	• Collecting critical data from external files • Running third party executables inserted in unusual files
Post Analysis Backdoor Generation	• Code that searches the server for particular file types to inject with backdoors

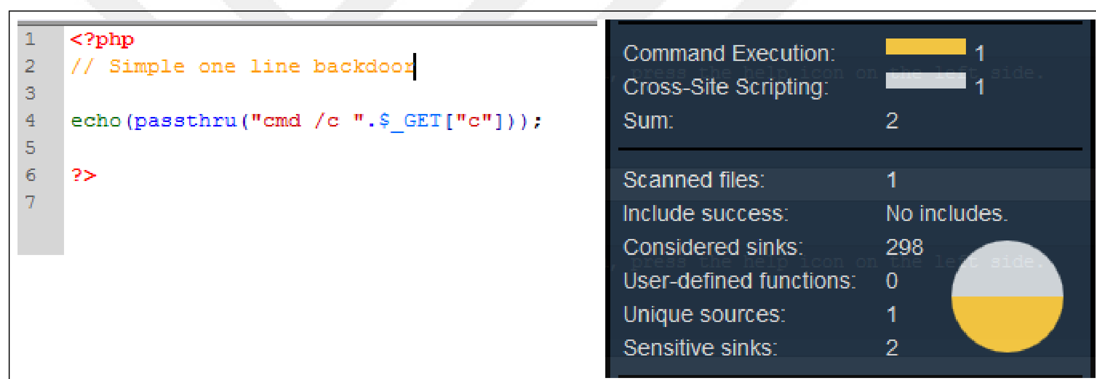

FIGURE 4.1: Result of static code analysis run byRIPS on a simple backdoor using passthru function with direct parameters.

same technique. In this case the parameter which is a shell command emerges during execution. The sample code in the figure below, at first sight, appears as a regular string operation. However, after conversion from hex to string, it turns out to be a command to be executed. The ch value is in fact 'start /d "path" file.exe'.

```
$ch = "0x730x740x610x720x74 0x2f0x64 0x220x700x610x740x680x22
0x660x690x6c0x650x2e0x650x780x65";
```

Encrypting the parameters is not the only way to hide them. In some backdoors, the key parameter is formed by various string operations such as picking letters from certain positions of a lengthy text. String splitting, keyword substitution and similar operations

Table 4.3: Summary of Backdoor Obfuscation Classification

Class of Obfuscation	Observed Methodologies in Backdoor Instances	Method of Encounter
Run Time Parameter Modification	• Encrypting functions • Sophisticated string operations such as keyword substitution, string splitting, character picking	Backdoor instances and reported cases
Intentional Vulnerabilities	• Deficiencies left on input filtering mechanisms • SQL and command injection • File inclusion	Backdoor instances and reported cases
Multistage Backdoor Insertion	• Software updates that transform the code into a backdoor	Reported cases
Hidden Logic Flow and High Code Complexity	• Alterations and additions on conditional branches	Backdoor instances and reported cases
Hiding Malicious Content on Indirect Data	• Collecting critical data from external files • Running third party executables inserted in unusual files	Backdoor instances and reported cases
Client Side Backdoor Obfuscation	• String manipulations of JavaScript code • Sending malicious code with PDF executables	Reported cases
Post Analysis Backdoor Generation	• Code that searches the server for particular file types to inject with backdoors	Backdoor instances
Embedding Malicious Binary to Software	• Injecting random binaries by using code caves in executables • Binary protecting methods of packer tools	Reported cases

are often performed on the parameters to make automated detection difficult.

```
$a1= "a}4era3?xqgclasd95sda";
$a2= "u756)&m!2sdh84Y%dg";
...
$functionParameter = $a1[11].$a2[6].$a3[10] ... ;
// $functionParameter goes as "cmd ... ";
```

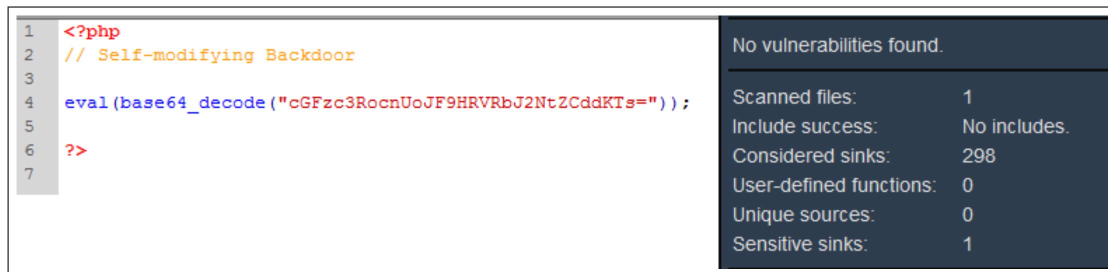

FIGURE 4.2: Result of static code analysis run by RIPS on a simple backdoor using passthru function with obfuscated parameters

The code in the figure above is a simplified illustration of forming a critical function parameter. The complexity of the operations will prevent the observer from predicting the value forged from a1 and a2 parameters. This formation of the parameters is particularly effective against keyword based detection techniques. Simply since, there is not any keyword on the code to detect until execution.

Computing parameters of the functions that insert special credentials into credential tables on run time is the most critical aspect of this technique. A username and password commonly added to the data structure that allows access to the attacker later in stage. In most cases of the backdoors examined, hardcoded usernames and passwords are almost instantly discovered even with naked eye. However, the transformation of the password and user strings creates an ambiguous case that is difficult to detect. Examples of backdoors using special credentials can be found at [8, 42].

4.2 Intentional Vulnerabilities Left on Software

Backdoors can emerge unintentionally as programming mistakes, or they can be planted in the software as an inside job which makes them deliberate backdoors. Attackers ordinarily try to diminish the risk of disclosure by disguising the deliberate backdoors as unintentional vulnerabilities. Oddly, this is not just a practice of masking intentions; it also helps the backdoor pass static analysis as the main focus of the vulnerability analysis will be a pre-determined vulnerability list and not the manipulations of logic and function.

The most common practice of an intentional vulnerability is the specific deficiencies left on input filtering mechanisms. A specific form of malicious input, which only the attacker knows, is allowed through the security mechanism that leads to compromise of

the system through several vulnerabilities such as SQL and command injection.
A reported example of intentional vulnerabilities is the removed backdoor on Linux 2.6 kernel, which can be found in Wysopal and Kernal Trap. The dynamic was created around the use of '=' operation instead of '=='. The attacker is able to call the function of this code and get the user root privileges.

```
if ((options == (__WCLONE|__WALL)) &&
(current->uid = 0))
retval = -EINVAL;
```

In the fight against the backdoors, enforcement of secure coding process is a strong weapon. Utilization of frameworks that have proven safe highly reduces the risk of mistakes and vulnerabilities remaining on the application. Therefore, the injection of specific vulnerabilities on parts of the software becomes much more difficult. A good practice of the frameworks is to force the functionalities depended on the critical functions to be practiced in a certain way that can be controlled and inspected.

4.3 Multistage Backdoor Insertion

Static analysis performed on the code are effective at finding the vulnerabilities, not predicting them. Therefore, not injecting the code with a backdoor at once but rather in multiple steps is an effective obfuscation technique. Multistage backdoor insertion makes it almost impossible to find the vulnerability at an early stage, as it does not exist. Later in a future stage regardless if it is a software update or a bug fix, most likely after the security inspections are performed, the backdoor is completed.
Most visible sign of this obfuscation is unreachable code blocks, even though it is not a certain indicator, a piece of code that is not reachable is immediately suspicious, it must be closely examined and preferably removed.

4.4 Hidden Logic Flow and High Code Complexity

Complexity of a software is crucial for the ability to understand what it performs. Studies suggest that if the cyclomatic complexity of single unit is over 50, the code is unstable

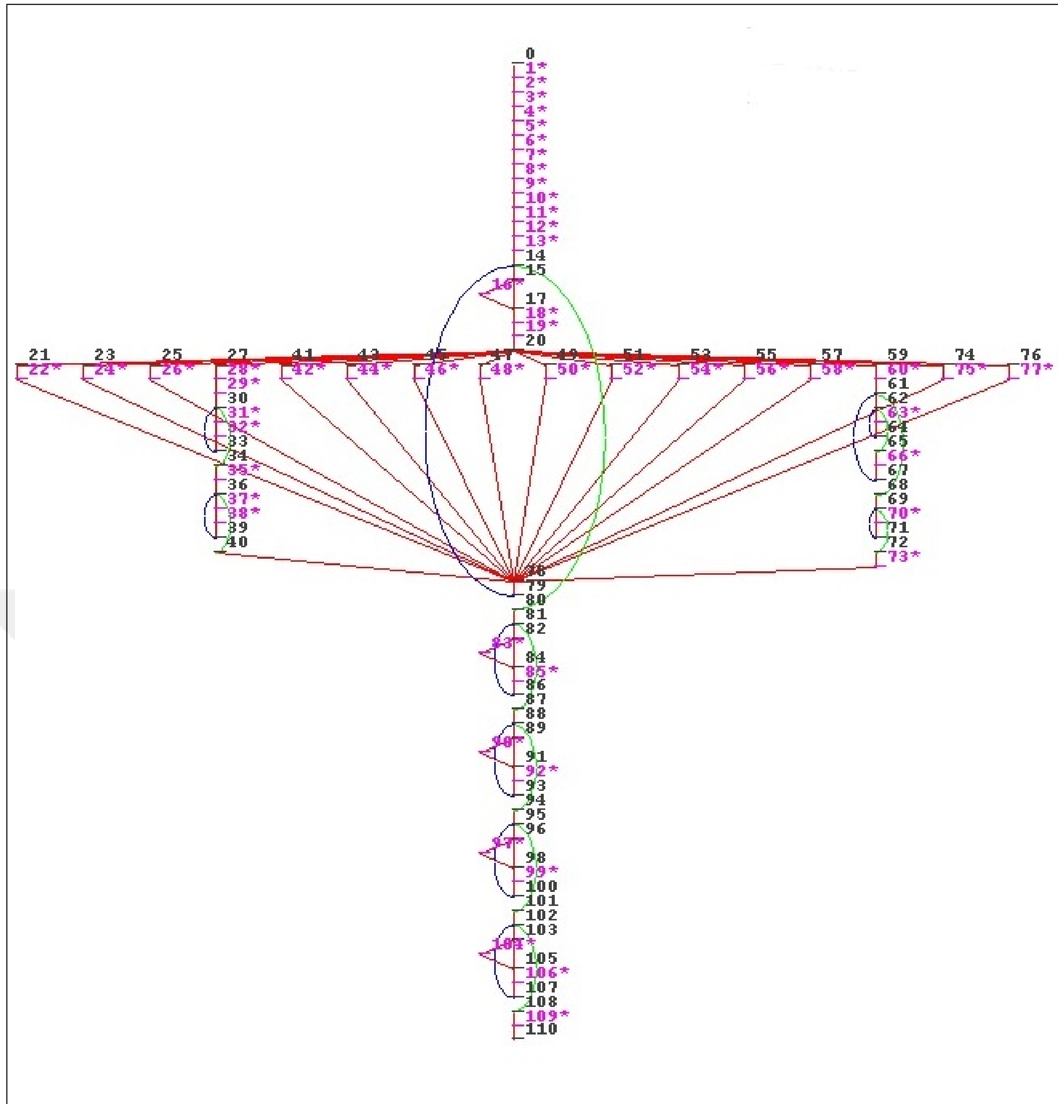

FIGURE 4.3: McCabe's Code Complexity Analysis Graphs - 1

and in very high risk [43]. In other words, it is very difficult to maintain and understand the logic of the code. Therefore, the chances of detecting a vulnerability is in jeopardy. Workflow analysis on complexity graphs as seen in Figure 4.3 and 4.4 helps understand the logic of a program. In fact, if there is an undesired side path that bypasses the security checks, it is to be spotted on these graphs. Nevertheless, the graphs need to be readable so that the backdoors can be detected. A complex software without proper design or testing comes out as a complex structure that is almost impossible to truly understand.

Even though, some studies suggest that there are only weak correlations between vulnerabilities and complexity measures [44], it is commonly accepted that the software

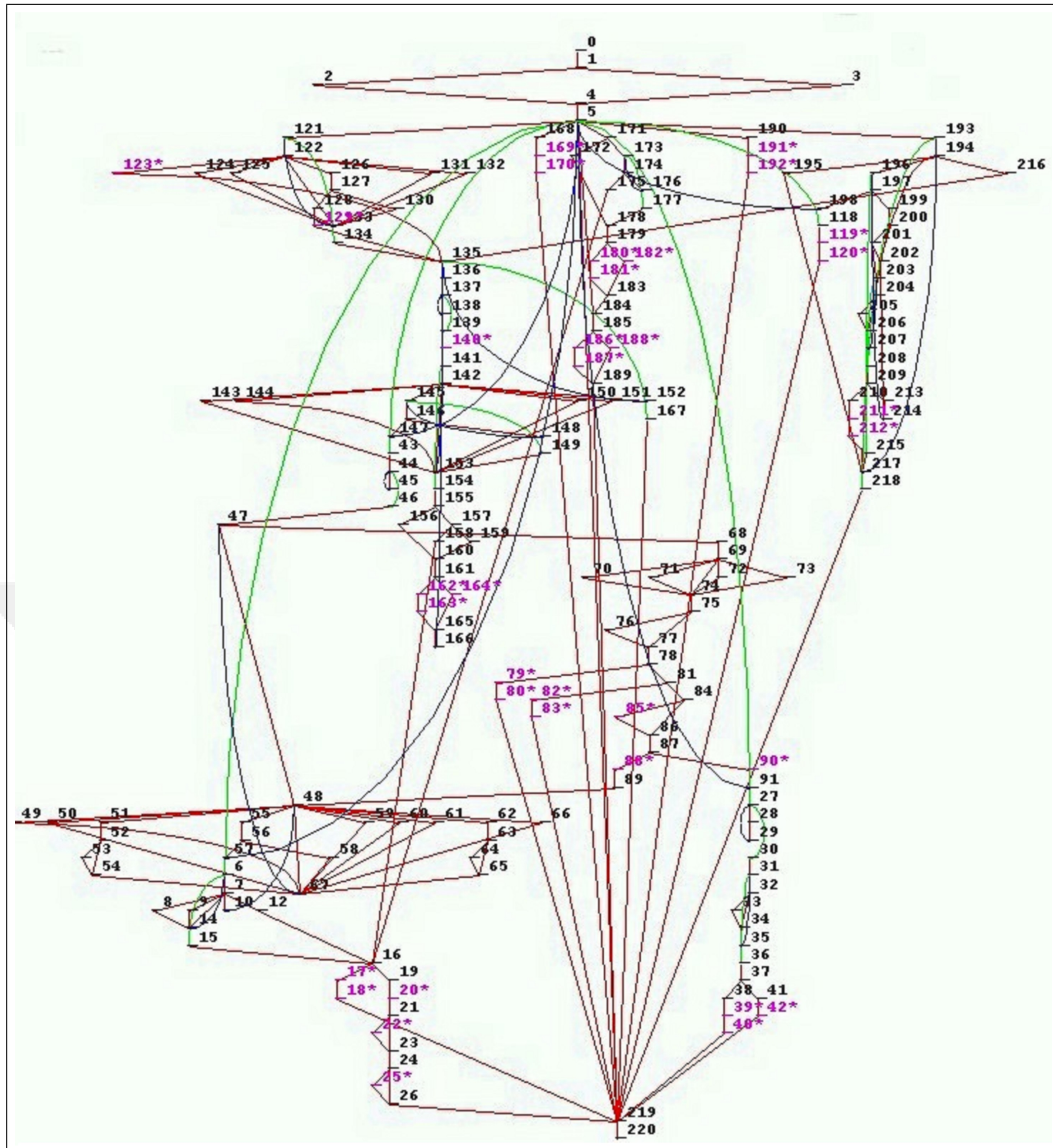

FIGURE 4.4: McCabe's Code Complexity Analysis Graphs - 2

complexity is an enemy of software security [45, 46]. Also, these studies are performed without the assumption that an attacker with ill intentions actively participated in the development of software.

In a backdoor detection centered point of view, the supervision of the code complexity has superior significance. In the trials we run on backdoor samples, tools such as pixy and RIPS failed to detect the vulnerability when the backdoor is obfuscated with complex structures, whereas they detected the vulnerabilities in the code when vulnerable piece of code is picked and replaced in a separate simple file.

The problem is not only the raw increase of the complexity, but the alterations and

additions on conditional branches that are directly connected on the malicious pieces of software. These techniques being combined with the common increase in the complexity yields software that cannot be analyzed.

4.5 Hiding Malicious Content on Indirect Data

It is a common technique to hide a malicious content not in the software but rather in the metadata of a file that does not go through investigative analysis. Most common instances of hidden contents are IP addresses, system commands, random looking chunks of data to be meaningful after combination with data from different sources.
One way to make this technique work is combining it with a vulnerability left on the software. The improper permissions of the malicious file for instance, will be giving it access to operate at the system level. Later the attacker who knows the file location and what it does can reach the file executing it with the information it contains. In this case the backdoor is a complete functional mechanism itself. Although in some cases, it is a small piece of a bigger mechanism such as a critical function parameter or a special credential that collected from the indirect data during operation.
A simple trick as giving the backdoor an unusual file extension also proves effective. If not set properly, the static code analyzers are tent to pass on unfamiliar file extensions leaving them unchecked. Therefore, it is crucial to know each file type in the source location along with their purpose. Nonfunctional, old and unnecessary files must be removed from server once the production begins.

4.6 Client Side Backdoor Obfuscation

Client side scripts are very convenient to hide backdoors. The dynamic features provided by languages such as JavaScript, when combined with string manipulations, can easily be turned into obfuscation routines [47]. Furthermore, there are public websites where you can easily obfuscate your JavaScript code with the click of a button.
Client side codes are interpreted and run at the client side, which is in general web browsers and similar applications that can provide the source of the script to the user. Therefore, many script developers feel the need to obfuscate their code in a way that is difficult for the users to copy and use in their applications. Hence, the obfuscation of

the code does not necessarily mean that the intention of the script is malicious, which makes script coding even a better target to inject with a backdoor.

Client side scripts are great targets for attackers, not only because it is difficult to detect them by static code analysis, also because it is difficult to detect them using anti-virus programs [47]. Therefore, a number of studies have been performed to detect backdoors in JavaScript code using various techniques, some of which are [20, 26, 47, 48]. However, the exploitation of the scripts are still very common and obfuscation techniques are becoming rapidly popular.

Consider the JavaScript code below which changes the attributes of a link that is not visible since it has no text value. By changing the ’href’ attribute the code specifies the download file url to be an exe file from an unknown website. It clearly intends to deceive the user to download a file that has malicious content.

```
// JavaScript Code
var re = document.getElementById('2');
re.href = "http://www.MaliciousSide/first.exe";
re.text = "Download Your Expense Report";
<!-- HTML Code
<a href="" download="expenses.pdf" id="2"></a>
```

This code is easy to be detected as a backdoor. However, with the very simple obfuscation performed, it can no longer be detected neither by static code analysis nor anti-virus programs.

```
// Obfuscated JavaScript Code
eval(function(p,a,c,k,e,d){e=function(c){return
c.toString(36)};if(!''.replace(/^/,String)){while(c--
){d[c.toString(a)]=k[c]||c.toString(a)}k=[function(e){return
d[e]}];e=function(){return'\\w+'};c=1};while(c--
){if(k[c]){p=p.replace(new
RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('6
0=7.5(\'2\');0.4="1://3.8/9.e";0.d="a b f
c";',16,16,'re|http||www|href|getElementById|var|document|Malic
iousSide|first|Download|Your|Report|text|exe|Expense'.split('|'),0,
{}))
```

4.7 Post Analysis Backdoor Generation

Backdoors can also be put in place by 3rd party attackers who have found a way of uploading files to servers in order to execute commands easily. More often than not they add multiple backdoors in different programming languages to increase the chances of at least one backdoor being executed. There are malicious codes on the web ready to be inserted into a server. Once they are in, they start searching through files on the system and then injecting the ones that are matching the file extensions that they desire. The code below is one example that is created in assembly to inject an obfuscated PHP backdoor into the files it reaches after a search operation performed.
The PHP backdoor script to be injected in this example is also heavily encoded. Once decoded from hexadecimal, it strikes as an eval code with a parameter encoded in base64. Furthermore, the base64 decryption is not performed directly on the text but a combination of text and ASCII translation on numbers to be added to the text, which makes it even harder to detect based on encryption of parameters.

```
"\x31\xc0" // xor %eax,%eax
"\x50" // push %eax
"\x83\xec\x01" // sub %esp,$0x1
...
/* PHP Backdoor to inject */

"\x65\x76\x61\x6c\x28\x62\x61\x73\x65\x36\x34\x5f\x64\x65\x63"
"\x6f\x64\x65\x28\x4d\x63\x6b\x78\x32\x2e\x63\x68\x72\x28\x34"
"\x37\x29\x2e\x66\x6a\x73\x4b\x54\x4e\x67\x44\x48\x4a\x4d\x64"
"\x74\x71\x52\x6c\x6a\x4e\x67\x44\x48\x62\x61\x68\x64\x59\x7a"
...
"\x34\x6e\x68\x73\x41\x76\x4e\x67\x41\x29\x29\x3b"
/* End payload ... */

"\x3c\x68" // cmp %al,$0x68
"\x74\x6d" // je short .me
...
```

These type of an operation, as it is useful for outside attackers, can be a tool for an insider that intends to inject the code with a backdoor. It guarantees that the source

code is clean at the time of static analysis, and then somehow triggers the process of changing the code with a distinct program. This is why security checks being cautiously performed for the servers is substantially significant to minimize the risk of having our code being injected at any later time. Any process that is not related to the service being provided by the web server should not be able to run. In this way, the chances will be better for preventing the attacker using sources on the server that is not checked by the security measures performed.

4.8 Embedding Malicious Binary to Software

In this study, we have focused on the obfuscation of vulnerabilities on the application source code and not on the compiled binary code. Nevertheless, the method of malicious binary injection is popular among attackers and must be considered along with the effective methods of detection.

The binary source generated after compilation of the source code can be injected with many creative methods. It is possible to embed Python code into another application using official Python documentation [49]. There are even open web instructions guiding users to injecting random binaries using debuggers and the code caves in executables. Furthermore, after the injection, the code can be obfuscated by packer tools and binary protecting methods.

Static and dynamic analysis methods are used on the executables to detect vulnerabilities. However backdoor detection on the binary code is a challenging task as there are over sophisticated methods designed to protect binary content. Making changes to the binary sources to hide software content is not specific to backdoors. In order to prevent malicious use of reverse engineering against binaries, various techniques have been developed called binary obfuscation [50].

A broad examination of the binary code obfuscation techniques used by the packer tools that are most popular with malware authors can be found at [51].

Chapter 5

Evaluation of Static Code Vulnerability Analyzers on Backdoors

In this section, results from the static analysis of the backdoor samples are evaluated. Results have been grouped based on the classification of the obfuscation techniques observed, with a separate group of backdoors that are without any obfuscation. Later a technique is suggested as a pre-analysis procedure to improve the detection rates, and the results of the analysis after the suggested technique is presented.

Table 5.1 and 5.2 gives the summary of the detection rates of the backdoors that are grouped based on the obfuscation classes. Figure 5.1 shows the detection rates of the tools both on obfuscated and not obfuscated backdoors in order to present the effects of the backdoor obfuscation on the detection rates for each tool. It must be noted that tools have reported the issues as possible vulnerabilities and not as detected backdoors, which leaves no room for false detection analysis and leads to higher detection rates in the results. However, same dynamic applies to the obfuscated backdoor analysis. Hence, the drop rates in the detections stay meaningful showing the effects of the obfuscation.

It is observed that 2 of the classes of obfuscation is not detected at all by any of the analyzers. The reason for the failure to detect hidden malicious content on indirect data is the tool incompetence to perform analysis on files with unknown file extensions. At the same time, the backdoor that is generating post analysis backdoors does contain an

encoded PHP backdoor that is not detected even when file type is converted into the suiting php extension.

The trials have shown that certain types of backdoor obfuscation classes can be disabled with particular strategies. These are, lowering the complexity of the logical structure, allowing unfamiliar file extensions, and checking whether an encrypted parameter is used in a critical function. These methods increase the detection rates for 3 of the obfuscation classes directly and contribute to the overall detection rates in process. A comparison of the detection rates of the samples, before and after the utilization of the disabling strategies, can be seen in Figure 5.2.

The first step of the procedure is to change the file extension of the unfamiliar file types to the extension of the analysis which allows the tools to check keywords and special functions hidden in the file. While the tools detect the direct usage of the code in the file, the step is finalized with the control of IP addresses and encrypted data in order to detect cases where the parameter of a special function is received. In both samples of the backdoors that are obfuscated with the hidden data on indirect data, this technique has been successful. It must be noted that this step has not been tested on a large number of unfamiliar file types with changed file extensions where complications such as the run time crush of the tools might occur.

The second step is lowering the cyclomatic complexity of the samples in order to give tools simpler files to analyze. The files with cyclomatic code complexity over 5 have been divided into 2 files where the cyclomatic complexity is measured again. All of the backdoors obfuscates with code complexity, with the exception of 1 ASP file, were PHP samples. Therefore, PHP Depend has been used for the measurements of the cyclomatic complexity and the measurement for the single ASP file has been performed manually.

Checking encrypted parameters in a critical function has also proven to be a very effective technique. When all the instances using encryption on parameters are marked as detected backdoors, the strategy has resulted with the full detection of the targeted backdoors. However, all the instances of the encryption usage in the samples of this study were with malicious intentions. This leaves room for false detection rates for the applications where the parameters of the critical functions actually need encryption.

Table 5.1: Static Detection Rates on PHP Backdoor Samples

Class of Obfuscation	Num of Samples	YASCA	RIPS	VCG	RATS
Post Analysis Backdoor Generation	1	0%	0%	0%	0%
Hidden Logic Flow and High Code Complexity	9	11.11%	44.44%	66.66%	66.66%
Run Time Parameter Modification	17	11.76%	47.05%	29.41%	58.82%
Hiding Malicious Content on Indirect Data	2	0%	0%	0%	0%
Intentional Vulnerabilities Left on Software	13	15.38%	84.61%	76.92%	69.23%
Not Obfuscated	126	18.25%	90.47%	68.25%	76.98%

Table 5.2: Static Detection Rates on ASP and Java Backdoor Samples

Class of Obfuscation	Num of Samples	VCG
Post Analysis Backdoor Generation	-	-
Hidden Logic Flow and High Code Complexity	1	0%
Run Time Parameter Modification	3	33.33%
Hiding Malicious Content on Indirect Data	-	-
Intentional Vulnerabilities Left on Software	4	75%
Not Obfuscated	34	38.23%

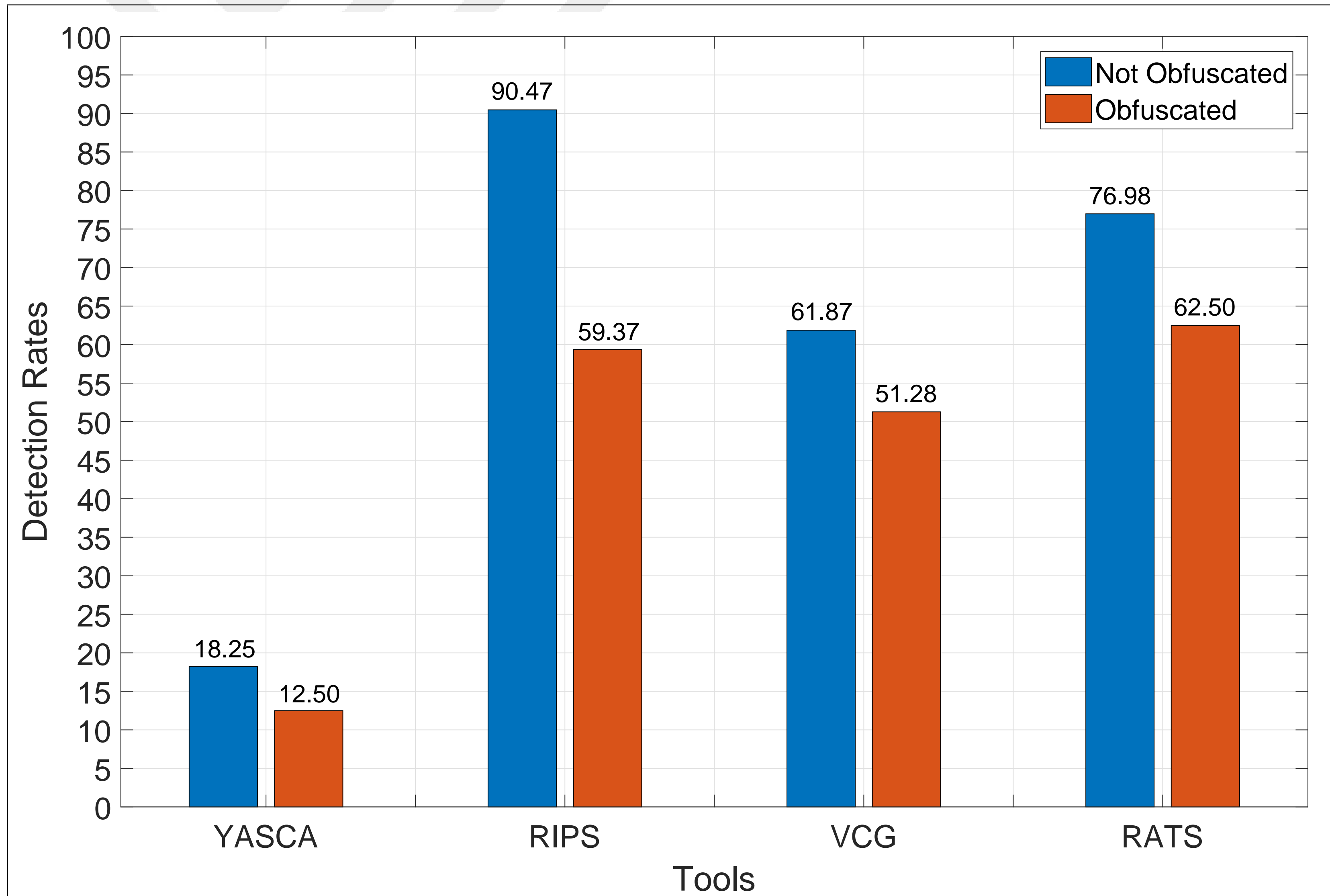

FIGURE 5.1: Detection rates of obfuscated and not obfuscated backdoors

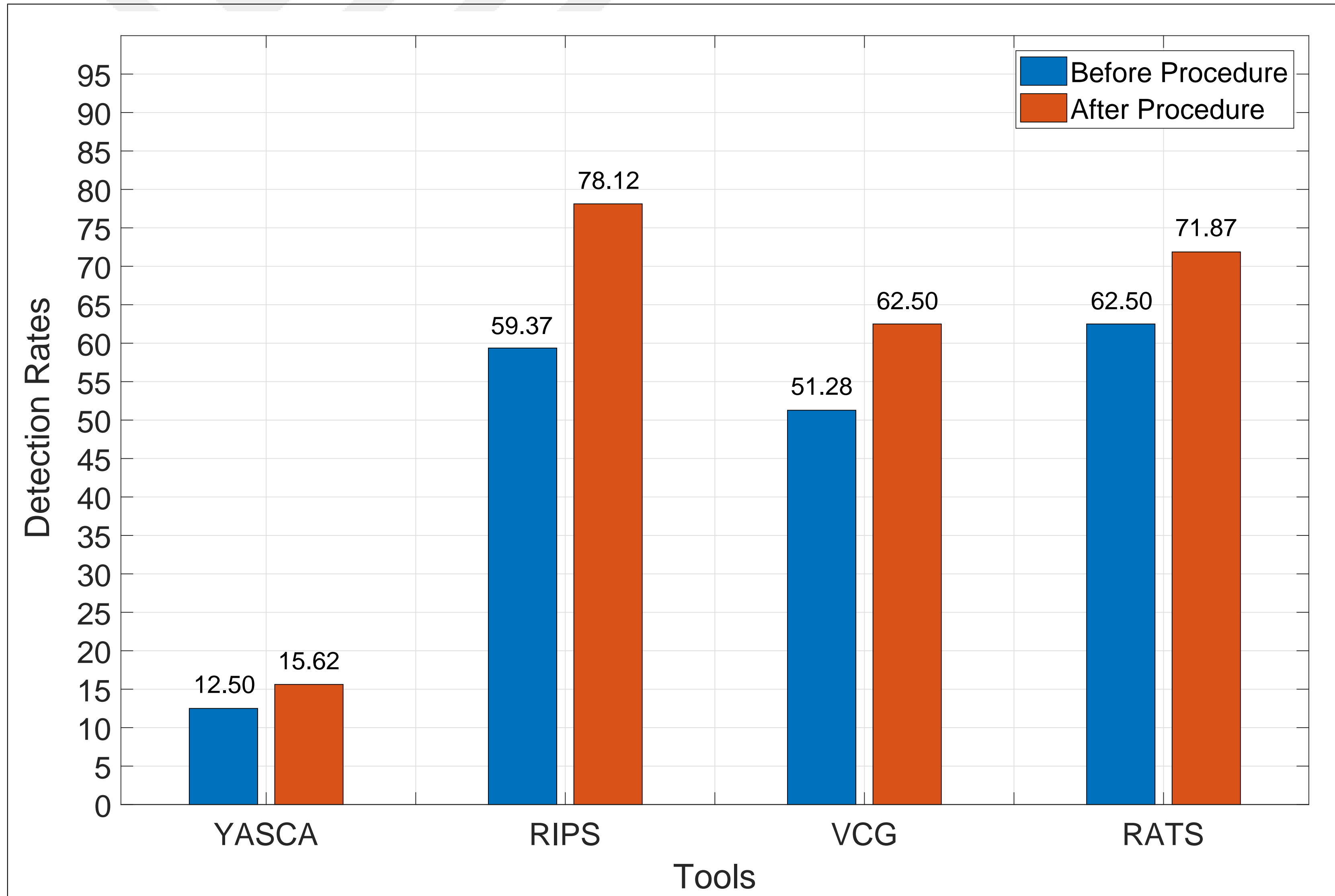

FIGURE 5.2: Comparison of the detection rates after the procedure

5.1 YASCA v2.2

Firstly, YASCA can contain other tools within itself. We have considered it as a core, without the external tools installed. Even though, YASCA is good at the exhibition of many software vulnerability issues, it has failed at the detection of the most cases of backdoors even without the obfuscations applied on them. For 159 PHP backdoor samples, it has given expected results on 27 samples with a lot of irrelevant issues. However, it clearly stated the unsafe practices of coding such as, weak authentication credentials and SQL injection, which is a strong feature against the intentional vulnerabilities left on the software.

5.2 RIPS v0.55

It is packed as a PHP code that can be put into the coding directory and accessed from a web browser. It has proven effective on most practices of backdoors such as command execution, file inclusion, and protocol injection. RIPS successfully detected expected vulnerabilities on 133 of 159 PHP samples. However, when the parameters of the functions are obfuscated with encoding, or the pieces of the code containing command execution are obfuscated with the addition of conditional branches, it failed on many cases.

5.3 Visual Code Grabber v2.1.0.0

Visual code grabber, which operates in all programming languages present in this study, has been successful on all cases of command injection backdoors of PHP, and the cases these injections are obfuscated by parameter modification. However, in the reports, these backdoors are not listed with high importance levels, meaning that the inspection of the case for whether the function has a malicious intent or not is left to the human review. Furthermore, the backdoor related vulnerabilities must be picked out from long extended list of vulnerabilities which are mostly minor issues. 102 samples of PHP backdoors have raised vulnerability issues in Visual Code Grabber. However, it could not show a similar performance with the cases of where backdoors are in Java and ASP. 17 out of 41 backdoors are detected in these languages using VCG.

5.4 RATS v2.3

It can be operated externally or within YASCA as an extension. RATS has given similar outcomes to Visual Code Grabber in terms of detected cases of PHP backdoors, listing them with low importance levels. RATS only support PHP language. 117 samples of PHP backdoors have raised vulnerability issues in RATS.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Obfuscation techniques on the web application backdoors are easy to apply and very effective on creating high complexity, thus making static analysis very difficult. The cost of an undetected backdoor in an application is disastrous in the means of information security and company reputation. Even worse than a regular software vulnerability since it raises questions for trust and incompetence. It is vitally significant to lower the risk of a backdoor during the development as much as possible. Therefore, creating a study ground on the backdoors and their obfuscation techniques is a promising work field.

In this thesis, we have presented a classification of obfuscation techniques applied on web application backdoors. Moreover, the performances of static code analyzers on backdoors and their obfuscations are examined.

On 200 instances of web application backdoors, and the reported cases of detection, 8 obfuscation techniques are identified. Only a limited number of open source static code vulnerability analyzers had notable effects on backdoors with obfuscation. Most of the tools were not even suited for cases where no obfuscation is applied.

The results show that there is a need for comprehensive research on putting forth effective static detection strategies specific to each obfuscation class in order to achieve a better positive detection rate.

6.2 Future Work

Current study on the backdoor obfuscation techniques has presented the state of the static code vulnerability analyzers. A next step to this study can be the close inspection of the methodologies to increase the success rate of the static detection on the samples and their obfuscation techniques. Detection strategies based on each obfuscation class in this study can be put forth, based on the observations where tools successfully detect backdoors and where they fail.

Bibliography

[1] S. Tzu. *The Art of War*. Classic bestseller. 2017. ISBN 9785000645390. URL https://books.google.com.tr/books?id=t8GjBAAAQBAJ.

[2] M. Apuzzo and M. S. Schmidt. Secret Back Door in Some U.S. Phones Sent Data to China, Analysts Say. The New York Times, Nov. 2016.

[3] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*, pages 297–300. IEEE, 2010.

[4] E. Konstantinou and S. Wolthusen. Metamorphic virus: Analysis and detection. *Royal Holloway University of London*, 15, 2008.

[5] U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, Aug 2006. ISSN 1772-9904. doi: 10.1007/s11416-006-0012-2. URL https://doi.org/10.1007/s11416-006-0012-2.

[6] Backdoor Definition. The Linux Information Project, Jan. 2006. Accessed: 2017-11-11.

[7] D. Mayers. Proceedings of advances in cryptology - crypto 96. 1996.

[8] C. Wysopal, C. Eng, and T. Shields. Static detection of application backdoors. *Datenschutz und Datensicherheit - DuD*, 34(3):149–155, 2010. ISSN 1862-2607. doi: 10.1007/s11623-010-0024-4. URL http://dx.doi.org/10.1007/s11623-010-0024-4.

[9] B. Schneier, M. Fredrikson, T. Kohno, and T. Ristenpart. Surreptitiously Weakening Cryptographic Systems. Cryptology ePrint Archive, Report 2015/097, 2015.

[10] J. Andrews. Linux: Kernel 'Back Door' Attempt. KernelTrap, Nov 2003.

[11] C. BarrÃa, D. Cordero, C. Cubillos, and M. Palma. Proposed classification of malware, based on obfuscation. In *2016 6th International Conference on Computers Communications and Control (ICCCC)*, pages 37–44, May 2016. doi: 10.1109/ICCCC.2016.7496735.

[12] The evolution of malware part one: 1949-1988. `https://www.tripwire.com/state-of-security/security-awareness/the-evolution-of-malware-part-one-1949-1988/`, Jun 2014. Accessed: 2017-11-07.

[13] J. Von Neumann and A. W. Burks. *Theory of self-reproducing automata*. University of Illinois Press Urbana, 1996.

[14] H. E. Petersen and R. Turn. System implications of information privacy. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 291–300, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465526. URL `http://doi.acm.org/10.1145/1465482.1465526`.

[15] M. E. Smid and D. K. Branstad. Data encryption standard: past and future. *Proceedings of the IEEE*, 76(5):550–559, May 1988. ISSN 0018-9219. doi: 10.1109/5.4441.

[16] P. Meissner. *Report of the Workshop on Estimation of Significant Advances in Computer Technology, Held at the National Bureau of Standards, August 30-31, 1976*. US Department of Commerce, National Bureau of Standards, 1976.

[17] S. Levy. Battle of the Clipper Chip. The New York Times, Jun. 1994.

[18] The 12 biggest, baddest, boldest software backdoors of all time. `http://www.infoworld.com/article/2606776/hacking/155947-Biggest-baddest-boldest-software-backdoors-of-all-time.html`. Accessed: 2017-04-15.

[19] K. Thompson. Acm turing award lectures. chapter Reflections on Trusting Trust. ACM, New York, NY, USA, 2007. ISBN 978-1-4503-1049-9. doi: 10.1145/1283920.1283940. URL `http://doi.acm.org/10.1145/1283920.1283940`.

[20] Y. Choi, T. Kim, S. Choi, and C. Lee. *Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis*, pages 160–172. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-10509-8. doi: 10.1007/978-3-642-10509-8_19. URL `http://dx.doi.org/10.1007/978-3-642-10509-8{_}19`.

[21] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–106. Springer, 2009.

[22] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium*, pages 169–186, 2009.

[23] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, pages 281–290. ACM, 2010.

[24] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 443–457. IEEE, 2012.

[25] Z. Li, Y. Tang, Y. Cao, V. Rastogi, Y. Chen, B. Liu, and C. Sbisa. Webshield: Enabling various web defense techniques without client side modifications. In *NDSS*, 2011.

[26] K. Schütt, M. Kloft, A. Bikadorov, and K. Rieck. Early detection of malicious behavior in javascript code. In *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, AISec '12, pages 15–24, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1664-4. doi: 10.1145/2381896.2381901. URL `http://doi.acm.org/10.1145/2381896.2381901`.

[27] S. M. Tabish, M. Z. Shafiq, and M. Farooq. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, CSI-KDD '09, pages 23–31, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-669-4. doi: 10.1145/1599272.1599278. URL `http://doi.acm.org/10.1145/1599272.1599278`.

[28] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001*, pages 38–49, 2001. doi: 10.1109/SECPRI.2001.924286.

[29] J. Z. Kolter and M. A. Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 470–478, New York, NY, USA, 2004. ACM. ISBN 1-58113-888-1. doi: 10.1145/1014052.1014105. URL http://doi.acm.org/10.1145/1014052.1014105.

[30] W.-J. Li, K. Wang, S. J. Stolfo, and B. Herzog. Fileprints: identifying file types by n-gram analysis. In *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, pages 64–71, June 2005. doi: 10.1109/IAW.2005.1495935.

[31] S. J. Stolfo, K. Wang, and W.-J. Li. *Towards Stealthy Malware Detection*, pages 231–249. Springer US, Boston, MA, 2007. ISBN 978-0-387-44599-1. doi: 10.1007/978-0-387-44599-1_11. URL https://doi.org/10.1007/978-0-387-44599-1_11.

[32] W.-J. Li, S. Stolfo, A. Stavrou, E. Androulaki, and A. D. Keromytis. A study of malcode-bearing documents. In *Proceedings of the 4th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '07, pages 231–250, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-73613-4. doi: 10.1007/978-3-540-73614-1_14. URL http://dx.doi.org/10.1007/978-3-540-73614-1_14.

[33] M. E. Saleh. *Detection and classification of obfuscated malware.* PhD thesis, 2016. URL https://search.proquest.com/docview/1793670651?accountid=17212. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2016-06-07.

[34] Microsoft Malware Classification Challenge (BIG 2015). https://www.kaggle.com/c/malware-classification, 2014. Accessed: 2017-11-11.

[35] Y. Mashevsky. New Malware Classification System. https://securelist.com/new-malware-classification-system/29875/, Nov 2004. Accessed: 2017-11-04.

[36] C. Bodnar. A Malware Classification. https://www.kaspersky.co.uk/blog/a-malware-classification/2620/, Oct 2013. Accessed: 2017-11-04.

[37] E. Gandotra, D. Bansal, and S. Sofat. Malware Analysis and Classification: A Survey. *Journal of Information Security*, 05(2), 2014. doi: 10.4236/jis.2014.52006. URL `www.scirp.org/journal/PaperInformation.aspx?PaperID=44440`.

[38] P. Szor. *The art of computer virus research and defense*. Pearson Education, 2005.

[39] Nist source code security analyzers. `https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html`. Accessed: 2017-04-15.

[40] Owasp source code analysis tools. `https://www.owasp.org/index.php/Source_Code_Analysis_Tools`. Accessed: 2017-04-15.

[41] Penetration testing labautomated source code review. `https://pentestlab.blog/2012/11/27/automated-source-code-review`. Accessed: 2017-04-15.

[42] Apc 9606 smartslot web/snmp management card backdoor. `http://www.securiteam.com/securitynews/5MP0E2AC0M.html`. Accessed: 2017-04-15.

[43] I. Heitlager, T. Kuipers, and J. Visser. A Practical Model for Measuring Maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pages 30–39, 2007. doi: 10.1109/QUATIC.2007.8.

[44] Y. Shin and L. Williams. Is complexity really the enemy of software security? In *Proceedings of the 4th ACM Workshop on Quality of Protection*, QoP '08, pages 47–50, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-321-1. doi: 10.1145/1456362.1456372. URL `http://doi.acm.org/10.1145/1456362.1456372`.

[45] D. Geer. A witness testimony in the hearing, wednesday 25 april 07, entitled addressing the nation's cybersecurity challenges: Reducing vulnerabilities requires strategic investment and immediate action. *submitted to the Subcommittee on Emerging Threats, Cybersecurity, and Science and Technology*, 2007.

[46] G. McGraw. *Software security: building security in*, volume 1. Addison-Wesley Professional, 2006.

[47] W. Xu, F. Zhang, and S. Zhu. Jstill: Mostly static detection of obfuscated malicious javascript code. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 117–128, New York, NY,

USA, 2013. ACM. ISBN 978-1-4503-1890-7. doi: 10.1145/2435349.2435364. URL `http://doi.acm.org/10.1145/2435349.2435364`.

[48] P. Likarish, E. Jung, and I. Jo. Obfuscated malicious javascript detection using classification techniques. In *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 47–54, 2009. doi: 10.1109/MALWARE.2009.5403020.

[49] Embedding python in another application. `https://docs.python.org/2/extending/embedding.html`. Accessed: 2017-04-15.

[50] B. Lee, Y. Kim, and J. Kim. binob+: A framework for potent and stealthy binary obfuscation. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 271–281, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-936-7. doi: 10.1145/1755688.1755722. URL `http://doi.acm.org/10.1145/1755688.1755722`.

[51] K. A. Roundy and B. P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.*, 46(1):4:1–4:32, July 2013. ISSN 0360-0300. doi: 10.1145/2522968.2522972. URL `http://doi.acm.org/10.1145/2522968.2522972`.