Detecting Derivative Malware Samples using Deobfuscation-assisted Similarity Analysis

Submitted in fulfilment
of the requirements for the degree of

Master of Science

of Rhodes University

Peter Mark Wrench

Grahamstown, South Africa
March 10, 2016

Abstract

The overwhelming popularity of PHP as a hosting platform has made it the language of choice for developers of Remote Access Trojans (RATs or web shells) and other malicious software. These shells are typically used to compromise and monetise web platforms by providing the attacker with basic remote access to the system, including file transfer, command execution, network reconnaissance, and database connectivity. Once infected, compromised systems can be used to defraud users by hosting phishing sites, performing Distributed Denial of Service attacks, or serving as anonymous platforms for sending spam or other malfeasance.

The vast majority of these threats are largely derivative, incorporating core capabilities found in more established RATs such as `c99` and `r57`. Authors of malicious software routinely produce new shell variants by modifying the behaviours of these ubiquitous RATs, either to add desired functionality or to avoid detection by signature-based detection systems. Once these modified shells are eventually identified (or additional functionality is required), the process of shell adaptation begins again. The end result of this iterative process is a web of separate but related shell variants, many of which are at least partially derived from one of the more popular and influential RATs.

In response to the problem outlined above, the author set out to design and implement a system capable of circumventing common obfuscation techniques and identifying derivative malware samples in a given collection. To begin with, a decoder component was developed to syntactically deobfuscate and normalise PHP code by detecting and reversing idiomatic obfuscation constructs, and to apply uniform formatting conventions to all system inputs. A unified malware analysis framework, called Viper, was then extended to create a modular similarity analysis system comprised of individual feature extraction modules, modules responsible for batch processing, a matrix module for comparing sample features, and two visualisation modules capable of generating visual representations of shell similarity.

The principal conclusion of the research was that the deobfuscation performed by the decoder component prior to analysis dramatically improved the observed levels of similarity between test samples. This in turn allowed the modular similarity analysis system to identify derivative clusters (or families) within a large collection of shells more accurately. Techniques for isolating and re-rendering these clusters were also developed and demonstrated to be effective at increasing the amount of detail available for evaluating the relative magnitudes of the relationships within each cluster.

Acknowledgements

During the course of this research, I was privileged to work with and enjoy the support of my supervisor, Professor Barry Irwin, without whose knowledge and guidance this project would never have reached completion.

I am also deeply and variously indebted to the following people: Dr Karen Bradshaw for her thorough editing, the Department of Computer Science at Rhodes University for the use of their excellent equipment and facilities, and to my family for their unwavering love and support.

Thanks must also go to the team behind the VirusTotal online analysis service for allowing me to source malware samples from their private API. Without these samples, the scope and impact of this research would have been greatly reduced.

Finally, I wish to acknowledge the financial support of Telkom, Tellabs, Stortech, Genband, Easttel, Bright Ideas 39 and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

Contents

List of Figures

List of Tables

List of Listings

1

Introduction

The overwhelming popularity of PHP as a hosting platform in recent years (Tatroe, 2005) has made it the language of choice for developers of web-based Remote Access Trojans (RATs or web shells) and other malicious software (Cholakov, 2008). These shells are typically used to compromise and monetise web platforms by providing the attacker with basic remote access to the system, including file transfer, command execution, network reconnaissance, and database connectivity. Once infected, compromised systems can be used to defraud users by hosting phishing sites, performing Distributed Denial of Service (DDOS) attacks, or serving as anonymous platforms for sending spam or other malfeasance (Landesman, 2007).

The proliferation of such malware has become increasingly aggressive in recent years, with some monitoring institutes registering over 390 000 new threats every day (AV Test, 2015). The vast majority of these threats are largely derivative, incorporating core capabilities found in more established RATs such as `c99` and `r57` (Kienzle and Elder, 2003; Slade, 2004). Authors of malicious software routinely produce new shell variants by modifying the behaviours of these ubiquitous RATs, either to add desired functionality or to avoid detection by signature-based detection systems (Collberg *et al.*, 1997; Sharif *et al.*, 2008b; Wang, 2001). Once these modified shells are eventually identified (or additional functionality is required), the process of shell adaptation begins again. The end result of this iterative process is a web of separate but related shell variants, many of which are at least partially derived from one of the more popular and influential RATs.

During the modification of the aforementioned shells, it has become common practice for malware authors to disguise their efforts by making extensive use of idiomatic obfuscation techniques designed to frustrate any efforts to dissect, modify, or reverse engineer the code (Kienzle and Elder, 2003; Wang, 2001). The resulting shells are functionally identical to their raw counterparts, but are more difficult to examine using static analytical techniques (Sharif *et al.*, 2008b). In order to determine the levels of similarity within a collection of shells and identify meaningful inter-sample relationships accurately, it is necessary to deobfuscate and normalise all inputs prior to analysis.

1.1 Problem Statement

The aim of this research was to create a system capable of accurately determining relationships between RATs written in PHP despite their use of idiomatic obfuscation constructs designed to thwart analysis attempts. The full problem statement is outlined below:

- PHP is widely used as a server-side scripting language, and as such is a popular choice for malware developers
- Many of the shells created by these developers are derivatives of seminal shells such as `c99` and `r57`
- When these shells are modified, authors often employ obfuscation techniques to increase their resilience to signature-based detection techniques
- The presence of these obfuscation constructs makes it difficult to identify relationships between derivative malware samples

1.2 Research Goals

In response to the problem statement described above, five research goals were identified:

1. The creation of a decoder component capable of normalising and deobfuscating test samples prior to similarity analysis. The purpose of this component would be to reverse commonly-used obfuscation idioms, thereby exposing more code for analysis.
2. The construction of four separate preprocessing modules designed to extract relevant features for comparison.
3. The implementation of a modular system designed to compare the features extracted by the preprocessing modules and create representative similarity matrices.

4. The creation of two visualisation modules capable of creating graphic representations of the results obtained during similarity analysis for ease of interpretation. The purpose of these modules is to facilitate the identification of meaningful relationships among samples by analysts.

5. An evaluation of the effects of the deobfuscation process on the results produced during similarity analysis.

1.3 Scope

It is important to note that the goal of this research was to identify derivative relationships between RATs in a collection of **known** malware. Although the tracking of malware variants could prove helpful as a method of detecting new web shells, the system was not designed as an antivirus solution for a production environment. Furthermore, the system specifically targets RATs written in PHP. As such, the approaches and techniques used during implementation explicitly target features of the PHP language, and would require extensive modification to be effective against malicious software developed using other languages. Finally, the deobfuscation and similarity analysis performed during this research involved the inspection of malware source files and not their binary forms, although many of the analytical approaches could be adapted for this purpose.

1.4 Previous Publications

Sections of the research presented in this thesis have previously been published as follows:

- Towards a Sandbox for the Deobfuscation and Dissection of PHP Malware. In Information Security for South Africa (Wrench and Irwin, 2014).
- A Sandbox-based Approach to the Deobfuscation and Dissection of PHP-based Malware. In South African Institute of Electrical Engineers African Research Journal, Volume 106 (Wrench and Irwin, 2015a).
- Towards a PHP Webshell Taxonomy using Deobfuscation-assisted Similarity Analysis. In Information Security for South Africa (Wrench and Irwin, 2015b).
- Detecting Derivative Malware Samples using Deobfuscation-assisted Similarity Analysis. In South African Institute of Electrical Engineers African Research Journal, In Press (Wrench and Irwin, 2016).

1.5 Thesis Structure

The remaining chapters of the thesis are organised as outlined below:

Chapter 2 introduces and discusses several important concepts related to code obfuscation and similarity analysis.

Chapter 3 outlines the design and implementation of the system created to identify meaningful inter-sample relationships.

Chapter 4 presents and critically evaluates the results obtained by the aforementioned system during the testing process.

Chapter 5 concludes the study by summarising the research and making suggestions for further research in the field.

2

Background and Previous Work

The detection of derivative malware samples is a non-trivial task with no well-defined solution. Many different techniques and approaches can be found in the literature, each with their own advantages and limitations. In an attempt to evaluate these approaches and contextualise the research, this chapter begins in Section 2.1 by providing an overview of the PHP language itself, including its notable features, performance relative to other languages, usefulness, inherent security characteristics, and most particularly its role as the language of choice for developers of RATs and other malware. Section 2.2 presents an overview of the structure and common capabilities of these RATs, as well as a discussion of the various methods of uploading them onto vulnerable hosts.

The concept of code obfuscation and the many methods of achieving it are discussed in Sections 2.3 and 2.4, before Section 2.5 goes on to discuss methods of reversing these techniques. Section 2.6 describes two idiomatic obfuscation constructs that are commonly employed by authors of PHP-based malware in particular. The VirusTotal online analysis framework that was used to source the majority of the test samples for this research is detailed in Section 2.7, while Sections 2.8 and 2.9 introduce the topics of lexical and similarity analysis, respectively. The Ssdeep approximate hashing tool is described in Section 2.10, followed by Section 2.11, which explains the three methods of data visualisation that were employed during this research. Sections 2.12 and 2.13 provide a discussion of previous work in the fields of code deobfuscation and similarity analysis, respectively. Section 2.14 concludes by summarising the key concepts and findings of the chapter.

2.1 PHP Overview

PHP[1] (the recursive acronym for PHP: Hypertext Preprocessor) is a general purpose scripting language that is primarily used for the development and maintenance of dynamic web pages. First conceived in 1994 by Rasmus Lerdof, the power and ease of use of PHP has enabled it to become the world's most popular server-side scripting language by numbers (Argerich, 2002). Using PHP, it is possible to transform traditional static web pages with predefined content into pages capable of displaying dynamic content based on a set of parameters. Although originally developed as a purely interpreted language, multiple compilers have since been developed for PHP, allowing it to function as a platform for standalone applications. Since 2001, the reference releases of PHP have been issued and managed by The PHP Group (Doyle, 2011).

2.1.1 Language Features

Much of the popularity of PHP can be attributed to its relatively shallow learning curve. Users familiar with the syntax of C++, C#, Java or Perl are able to gain an understanding of PHP with ease, as many of the basic programming constructs have been adapted from these C-style languages (Argerich, 2002; PHP Group, 2015a). As is the case with more recent derivatives of C, users need not concern themselves with memory or pointer management, both of which are dealt with by the PHP interpreter (McLaughlin, 2012). The documentation provided by the PHP Group is concise and comprehensively describes the many built-in functions that are included in the language's core distribution (PHP Group, 2015c). The simple syntax, recognisable programming constructs and thorough documentation combine to allow even novice programmers to become reasonably proficient in a short space of time (Argerich, 2002).

PHP is compatible with a vast number of platforms, including all variants of UNIX, Windows, Solaris, OpenBSD and Mac OS X (Argerich, 2002). Although most commonly used in conjunction with the Apache web server, PHP also supports a variety of other servers, such as the Common Gateway Interface, Microsoft's Internet Information Services, Netscape iPlanet and Java servlet engines (PHP Group, 2015d). Its core libraries provide functionality for string manipulation, database and network connectivity, and file system support (Argerich, 2002; Doyle, 2011; PHP Group, 2015j), giving PHP unparalleled flexibility in terms of deployment and operation.

As an open-source language, PHP can be modified to suit the developer. In an effort to ensure stability and uniformity, however, reference implementations of the language are periodically released by The PHP Group (Doyle, 2011). This rapid development cycle ensures that bug fixes and additional functionality are readily available and has contributed directly to PHP's reputation as one of the most widely supported open source languages in circulation today

[1] https://www.php.net/

(Argerich, 2002; Sklar, 2008). An abundance of code samples and programming resources exist on the Internet in addition to the vendor-provided standard documentation (The Resource Index Online Network, 2005; PHP Group, 2015e; Zend Technologies, 2013), while many extensions have been created and published by third party developers (PHP Group, 2015f).

2.1.2 Performance and Use

PHP is commonly deployed as part of the Linux, Apache, MySQL and PHP/Perl/Python (LAMP) stack (Bughin *et al.*, 2008). It is a server-side scripting language in that the PHP code embedded in a page will be executed by the interpreter on the server before that page is served to the client (Doyle, 2011). This means that it is not possible for a client to know what PHP code has been executed – they are only able to see the result. The purpose of this preprocessing is to allow for the creation of dynamic pages that can be customised and served to clients on the fly (Argerich, 2002).

When implemented as an interpreted language, studies have found that PHP is noticeably slower than compiled languages such as Java and C (Wu *et al.*, 2000; Titchkosky *et al.*, 2003). However, since version 4, PHP code has been compiled into bytecode that is subsequently executed by the Zend Engine, dramatically increasing efficiency and allowing PHP to outperform code written in other languages (such as Axis2 and the Java Servlets Package) (Cecchet *et al.*, 2003; Suzumura *et al.*, 2008; Trent *et al.*, 2008). Performance can be further enhanced by deploying commonly-used PHP scripts as executable files, eliminating the need to recompile them each time they are run (Atkinson and Suraski, 2004).

At the time of writing, PHP was being used as the primary server-side scripting language by over 240 million websites (Ide, 2015), with its core module, mod_php, logging the most downloads of any Apache Hypertext Transfer Protocol (HTTP) module (PHP Group, 2015i). Of the websites that disclosed their scripting language (several chose not to for security reasons), 79.8% were running some implementation of PHP, including popular sites such as Facebook, Baidu, Wikipedia and Wordpress (Web Technology Surveys, 2013).

2.1.3 Security

A study of the United States National Vulnerability Database performed in April 2013 found that approximately 30% of all reported vulnerabilities were related to PHP (Coelho, 2013). Although this figure might seem alarmingly high, it is important to note that most of these vulnerabilities are not vulnerabilities associated with the language itself, but are rather the result of poor programming practices employed by PHP developers. In 2008, for example, a mere 19 core PHP vulnerabilities were discovered, along with just four in the language's libraries (Coelho, 2013).

These numbers represent a small percentage of the 2218 total vulnerabilities reported in the same year.

Apart from a lack of knowledge and caution on the part of PHP developers, the most plausible explanation for the large number of vulnerabilities involving PHP is that the language is specifically being targeted by hackers. Because of its popularity, any exploit targeting PHP can potentially be used to compromise a multitude of other systems running the same language implementation (Coelho, 2013). PHP bugs are thus highly sought after because of the high pay-off associated with their discovery. This mentality is clearly demonstrated in the recent spate of exploits targeting open source PHP-based Content Management Systems like phpBB, PostNuke, Mambo, Drupal and Joomla, the last of which has over 30 million registered users (Miller, 2006; Open Source Matters, 2013).

2.2 Remote Access Trojans

Trojans horses are malicious scripts that claim to be desirable programs (Symantec Corporation, 2015). Distinguishable from true viruses by their lack of self-replicating mechanisms, these scripts often create backdoors that allow a remote user to gain persistent administrative control over a Personal Computer (PC) or server (Kazanciyan, 2012; Kienzle and Elder, 2003). Trojan horses that incorporate this functionality are alternately termed Remote Access Trojans (usually when installed on PCs) or web shells (usually when uploaded to production servers) (Kienzle and Elder, 2003). The terms are used interchangeably for the purposes of this research.

2.2.1 Purpose and Use

Modern web shells can be used either as part of a larger compromise attempt or as a standalone exploitation vector (Pfleeger and Pfleeger, 2002). They are most commonly used for the following purposes (Chuvakin, 2003; United States Computer Emergency Readiness Team, 2015):

- To access and exfiltrate sensitive data such as user credentials or financial information
- To act as a relay point between an attacker and the hosts inside the network without direct access to the Internet
- To create and maintain a persistent command-and-control structure such as a botnet
- To facilitate the uploading of additional forms of malware

2.2.2 Language and Structure

Web shells can be written in any language that is supported by the target web server. For this reason, many of the popular RATs are written in well-supported languages such as PHP, Active Server Pages (ASP), Python, Perl, and Ruby (Pfleeger and Pfleeger, 2002; United States Computer Emergency Readiness Team, 2015). Shell sizes can vary from extremely small samples containing just a single line of code to large, fully-featured samples consisting of thousands of lines (Hutchins *et al.*, 2011). Larger shells are generally self-sufficient, incorporating all the functionality deemed necessary by the author, while smaller shells often rely on external scripts or actions for effective exploitation.

Listing 2.1 contains an example of a rudimentary web shell written in PHP. Once uploaded onto the target server, this script simply executes the contents of the 'x' parameter as PHP code using the `eval()` function discussed in Section 2.6. An example of how this would be achieved is shown in Listing 2.2.

```
1 <?php
2   eval($_GET["x"]);
3 ?>
```

Listing 2.1: Simple PHP web shell

```
http://target.com/example.php?x=echo("owned")%3B
```

Listing 2.2: URL targeting the shell in Listing 2.1

Unlike the rudimentary example in Listing 2.1 that requires users to submit their malicious code via simple HTTP requests, many of the more advanced web shells include large sections of embedded HyperText Markup Language (HTML) that are used to create comprehensive and user-friendly Graphical User Interfaces (GUIs) (Haagman and Ghavalas, 2005). These simple GUIs enable users with limited technical expertise to access advanced functionality, as is demonstrated by the interface shown in Figure 2.1. The sample in question is a derivative of the popular `c99` shell, and includes interfaces that allow users to either execute predefined commands or upload their own. In addition to this, this particular variant provides simple interfaces for common I/O operations such as file creation and modification.

2.2.3 Common Capabilities

Although the capabilities of web shells vary according to their intended function, most shells provide basic access to the local file system. Remote users can add, modify, and delete files

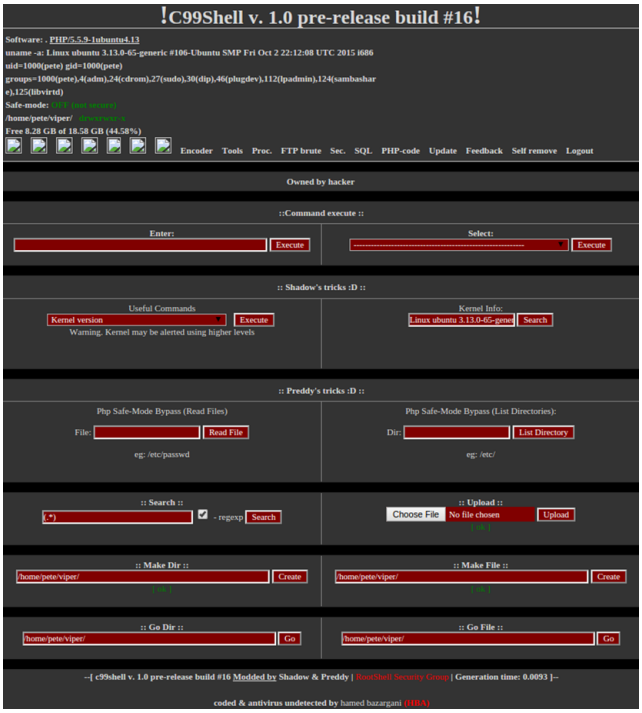


Figure 2.1: Interface of a derivative of the popular `c99` shell

as though they had physical access to the server. In addition to this, many more of the more developed samples provide additional functionality, some examples of which are provided below (Landesman, 2007; Decloedt and van Heerden, 2010; Hutchins *et al.*, 2011):

- Execution of arbitrary PHP scripts
- Keystroke logging
- The ability to include both local and remote files via FTP
- Automated detection of password files
- Registry editing
- Brute force password crackers
- Packet sniffing
- The ability to execute commands on both Linux and Windows systems
- Access to file search functionality
- The ability to send email from a compromised host
- Port binding functionality to allow for back connects and persistent access to specified folders

2.2.4 Delivery Vectors

Any server that supports the uploading of client-supplied content (usually via the HTTP POST method or compromised File Transfer Protocol (FTP)) is vulnerable to web shell infection (Decloedt and van Heerden, 2010). In addition to this, RATs can be delivered via a number of web application exploits and configuration weaknesses, including but not limited to the following (Katz, 2009; United States Computer Emergency Readiness Team, 2015):

- Structured Query Language (SQL) injection
- Cross-site scripting (XSS)
- Local and Remote File Inclusion vulnerabilities (LFI/RFI)
- Application vulnerabilities in Content Management Systems (CMSs)such as Joomla, Wordpress, and Drupal
- Compromised administration interfaces that provide file upload capabilities

2.3 Code Obfuscation

Code obfuscation is a program transformation intended to thwart reverse engineering attempts. The resulting program should be functionally identical to the original, but may produce additional side effects in an attempt to disguise its true nature.

In their seminal work detailing the taxonomy of obfuscation transforms, Collberg *et al.* (1997) define code obfuscation as a "potent transformation that preserves the observable behaviour of programs". The concept of "observable behaviour" is defined as behaviour that can be observed by the user, and deliberately excludes the distracting side effects mentioned above, provided that they are not discernible during normal execution. A transformation can be classified as potent if it produces code that is more complex than the original.

All methods of code obfuscation can be evaluated according to three metrics (Borello and Mé, 2008):

- Potency – the extent to which the obfuscated code is able to confuse a human reader
- Resilience – the level of resistance to automated deobfuscation techniques
- Cost – the amount of overhead that is added to the program as a result of the transformation

Although primarily used by authors of legitimate software as a method of protecting technical secrets, code obfuscation is also employed by malware authors to hide their malicious code. Reverse engineering obfuscated malware can be tedious, as the obfuscation process complicates the instruction sequences, disrupts the control flow and makes the algorithms difficult to understand. Manual deobfuscation in particular is so time-consuming and error-prone that it is often not worth the effort.

2.4 Methods of Obfuscation

Although the number of code obfuscation methods is limited only by the creativity of the obfuscator, the ones listed in the sections below fall neatly into the three categories of layout, data and control obfuscation (Linn and Debray, 2003; Li *et al.*, 2009). Each category boasts methods of varying potency, and a powerful obfuscator should employ methods from each category to achieve a high level of obfuscation.

2.4.1 Layout Obfuscation

Perhaps the most trivial form of obfuscation, layout obfuscation is concerned with the modification of the formatting and naming information in a program (Ertaul and Venkatesh, 2004).

Format Modification

The removal of formatting information such as line breaks and white space from source code is the most common method of obfuscation. It can only be performed on programs written in languages that do not depend on formatting as a structural device and is of low potency, as it removes very little semantic content and is easily processed by automated deobfuscation systems. This method is resilient to manual deobfuscation owing to the decrease in code readability, however, and can be performed without adding any overhead to the original program (Collberg *et al.*, 1997).

Identifier Name Modification

The transformation or scrambling of meaningful variable names into arbitrary identifiers is another common method of obfuscation (Ertaul and Venkatesh, 2004). Like format modification, it does not affect the efficiency of the resulting program (it contributes no additional overhead) and fails to confound automated deobfuscation systems (You and Yim, 2010). It is of a slightly higher potency, however, as variable names in unmodified form contain a wealth of semantic information that could be of use to a manual deobfuscator (Ertaul and Venkatesh, 2004).

2.4.2 Data Obfuscation

The obscuring of data structures in a program by modifying how they are stored, accessed, grouped and ordered is known as data obfuscation (Collberg *et al.*, 1997). It is considered more powerful than layout obfuscation as it obscures the semantics of a program and is able to stymie some automated deobfuscation systems (Sharif *et al.*, 2008b). Programs written using object-oriented languages in particular store much of their semantic information in the form of data structures. Data obfuscation is thus of paramount importance when attempting to obscure code written in these languages.

Storage and Encoding Modification

Modifying the data storage characteristics of a program changes the way data structures are stored in memory (Ertaul and Venkatesh, 2004). Typical examples of this type of obfuscation include variable splitting (parts of a single variable stored in many different locations) and the conversion of static data (such as a string) to procedural data (such as a function that produces the same string at runtime). The former makes it difficult to discern the purpose of a variable (it could be a variable fragment with no individual value) while the latter removes static data that may contain information that could be used to aid in the reverse engineering process (Linn and Debray, 2003).

Data Aggregation

Modifications to the way data are grouped in a program can also serve to obscure the data structures contained therein (Li *et al.*, 2009). Three common examples of this type of obfuscation are listed below:

- Scalar variables such as integers can be merged into a single variable provided that the single variable is sufficiently large to accommodate the scalar variables with no loss in precision. It is possible, for example, to store two 32-bit integers in one 64-bit integer, although this would then require major changes to how each variable is referenced in the rest of the program.
- Structures such as arrays can be merged, split, folded or flattened to increase their complexity. These techniques all complicate access to the arrays and further remove them from the data they are intended to represent (flattening a two-dimensional array that was intended to represent a chess board, for example, will make it more difficult to extract this representation during the obfuscation process).

- Class inheritance relationships can be complicated by splitting a single class into multiple classes or by introducing fake classes into the inheritance hierarchy. The result of these operations is a class structure in which classes no longer represent complete entities and relationships are convoluted and illogical.

Data Ordering

When constructing a program, it is common practice to follow the principle of locality of reference and group data structures with the functions that are likely to modify them (Denning, 2005). This tendency can be used by deobfuscators to identify which data structures are related to various functions, making it simpler for them to reverse engineer the code. Reordering data structures removes this advantage and increases the complexity of the deobfuscation process. Simple techniques include reordering variables (this often includes making some local variables global to thwart locality analysis), reordering object methods and their parameters, and reordering elements within an array (You and Yim, 2010). When data reordering is combined with data aggregation and storage, and encoding modification, it becomes very difficult for a deobfuscator to correctly restore the program's data structures (Collberg *et al.*, 1997).

2.4.3 Control Obfuscation

Perhaps the most important characteristic of a program that needs to be obscured during the obfuscation process is the control flow. Reverse engineering a program when the control flow and data structures are known is a trivial process – as previously discussed, other obfuscation methods such as layout modification are simple to overcome. As is the case with the obfuscation of data, the aggregation and ordering of control flow statements are important and can be modified to increase the program's complexity and resilience (You and Yim, 2010).

Computation Modification

The modification of the computations involved in the determination of control flow (such as condition calculations in loops and predicate evaluation in if statements) is a powerful method of obfuscation, although it does introduce a significant amount of overhead into the resulting program (Collberg *et al.*, 1997). Computation modification can be achieved in the following ways:

- Irrelevant code (i.e., code that has no impact on the control flow) can be inserted into a program to frustrate deobfuscators and make the reverse engineering process more time-consuming, as the deobfuscator has no way of knowing whether a section of code is irrelevant until it has been processed (Ertaul and Venkatesh, 2004).

- Dummy processes can be added to the program to distract reverse engineering attempts and code can be parallelised to complicate the control flow, making it more difficult to unravel (Li *et al.*, 2009). The latter technique is considered one of the more powerful methods of obfuscation, as each parallel process increases the number of possible execution paths exponentially, greatly complicating and sometimes defeating the deobfuscation process altogether.

Code Aggregation

Much like data aggregation, code aggregation merges dissimilar blocks of code and separates similar blocks of code. Collberg *et al.* (1997) describe the twin goals of code aggregation as follows:

- Code that a programmer has placed in a method (because it logically belonged together) should be scattered throughout the program
- Code that has no logical relationship should be aggregated into a single method

Further obscuring of the abstractions usually employed by programmers can be achieved through the use of inline and outline methods (Linn and Debray, 2003). Instead of abstracting commonly used code into a separate method, an obfuscator will include this code (as an inline method) wherever it is needed, effectively removing a semantically rich procedural abstraction that could be leveraged by a deobfuscator. Outline methods, by contrast, abstract a section of code that is not commonly used into a separate method, granting it undeserved status as a procedural abstraction and potentially misleading any reverse engineering attempts (Collberg *et al.*, 1997).

Code Ordering

When writing code, programmers tend to organise expressions and statements in a logical manner that makes the program easy to read and understand. Since the goal of obfuscation is to discourage understanding, it follows that the ordering of code should be as random as possible. This is trivial for structures such as methods in classes, but in some cases the ordering of statements cannot be entirely randomised because of the dependencies that exist between them (a variable declaration cannot be placed below an expression that includes that variable, for example). In these cases, a dependency analysis of the two statements must be performed before any form of code reordering is attempted. Although reordering is not a powerful method of obfuscation when used in isolation, its effectiveness increases when combined with code aggregation and computation modification. (Ertaul and Venkatesh, 2004)

2.5 Deobfuscation Techniques

The obfuscation methods described in the previous sections are all designed to prevent code from being reverse engineered. Given enough time and resources, however, a determined deobfuscator will always be able to restore the code to its original state. This is because perfect obfuscation is provably impossible, as is demonstrated by Barak *et al.* (2001) in their seminal paper "On the (Im)possibility of Obfuscating Programs". Collberg *et al.* (1997) concur, postulating that every method of code obfuscation simply "embeds a bogus program within a real program" and that an obfuscated program essentially consists of "a real program which performs a useful task and a bogus program that computes useless information". Bearing this in mind, it is useful to review the techniques that are widely employed by existing deobfuscation systems.

2.5.1 Pattern Matching

Sophisticated deobfuscation systems are able to construct databases of previously detected bogus code segments. They can then compare fragments of an obfuscated piece of code with the patterns stored in the database and remove these fragments from the program before applying the other techniques described below. The resultant decrease in the size of the program greatly increases the efficiency of the deobfuscator – the larger the database is, the greater is the increase in efficiency (You and Yim, 2010).

2.5.2 Program Slicing

Deobfuscators that employ program slicing techniques are able to split an obfuscated program into manageable units called slices that can then be evaluated both individually and in relation to other slices. In this way, the system can avoid bogus code entirely and group similar code blocks together, reversing the efforts of the obfuscator and making the code more readable. Advanced slicing systems are able to create chains of slices leading up to a target slice that represent the code blocks that were executed up to that point, even if said blocks are scattered throughout the program (Collberg *et al.*, 1997).

2.5.3 Statistical Analysis

Like pattern matching, statistical analysis aims to remove unimportant code, but it is able to do so without knowledge of previously discovered bogus segments (Li *et al.*, 2009). Instead, the deobfuscator will repeatedly test an expression in an obfuscated program and record the results (Sharif *et al.*, 2008a). If the expression always returns the same value, it is likely to belong to the meaningless part of the obfuscated code and can safely be replaced with the value itself or removed from the program altogether.

2.5.4 Partial Evaluation

A partial evaluator is a system capable of splitting a source program into a static segment and a dynamic segment. The static segment consists of all the code that can be identified and computed by the evaluator prior to runtime. This code can be considered unimportant in the sense that it produces no useful result and therefore corresponds to the spurious code blocks often introduced by code obfuscators. Once the static segment has been removed, the remaining dynamic segment represents the original program (You and Yim, 2010).

2.6 Code Obfuscation and PHP

As a procedural language with object-oriented features, PHP can be obfuscated using all of the methods detailed in Section 2.4 (PHP Group, 2015k). In practice, however, two built-in code execution functions account for the majority of code hiding efforts and are specifically marked by the PHP Group as being potentially exploitable (Wrench and Irwin, 2014; PHP Group, 2015b,g).

As a result of its ability to execute an arbitrary string as PHP code, the `eval()` function is widely used as a method of hiding code. The potential for exploitation is so great that the PHP Group includes a warning against its use, advising that it only be used in controlled situations, and that user-supplied data be strictly validated before being passed to the function. (PHP Group, 2015b)

The `eval()` function is often combined with auxiliary string manipulation functions to form the following obfuscation idiom (Wrench and Irwin, 2014):

```
eval(gzinflate(base64_decode('GSJ+S...')));
```

The string containing the malicious code is compressed before being encoded in base64. At runtime, the process is reversed. The code that is produced is then executed through the use of the `eval()` function.

The `preg_replace()` function is used to perform a regular expression search and replace in PHP (PHP Group, 2015g). Although this does not present a problem in itself, the deprecated '/e' modifier allows the resultant text to be executed as PHP code (in effect causing an `eval()` function to be applied to the result). An example of the use of the `preg_replace()` function for hiding code is shown in the following code extract:

The example shows a very simple `preg_replace()` function that searches for the pattern `'x'` in the string `'y'`, replaces it with the string `'echo($a);'` and then evaluates the resulting

```
preg_replace('/x/e', 'echo($a);', 'y');
```

code. In this case, the text contained in the `$a` variable would be displayed if the code was executed.

2.7 VirusTotal

Originally developed by Hispasec in 2004, VirusTotal[2] is a Google-owned online analysis service that can be used to scan files for malicious content (VirusTotal Team, 2015a). The company's stated vision is to "improve the antivirus and security industry and make the Internet a safer place through the development of free tools and services" (VirusTotal Team, 2015a). In pursuit of this goal, the service aggregates results produced by a total of 55 commercial antivirus products when run against a given file (Sanz *et al.*, 2013). A full list of these antivirus engines is provided in Appendix A.

Files suspected of containing malicious code can be uploaded to VirusTotal in three ways: via the web interface, by email, or through scripted submissions to the service's online Application Program Interfaces (APIs), which are discussed in greater detail in Sections 2.7.1 and 2.7.2 (VirusTotal Team, 2015a). Once the analysis has been successfully completed, the service returns a JSON object containing the results produced by each scanning engine, including detection labels that attempt to identify the type of malware under investigation. Of particular interest to this research is the "Backdoor:PHP" designation assigned to appropriate samples by Microsoft's Malware Protection engine, which was used to isolate and retrieve RATs written in PHP for use as inputs during system testing.

2.7.1 Public API

VirusTotal's public API is a free service that allows client applications to upload and scan files, Uniform Resource Locators (URLs), and Internet Protocol (IP) addresses (VirusTotal Team, 2015c). Users with a public key also have access to reports relating to all previous scans performed by the service, and can create automated sample comments within VirusTotal's research community.

The primary method of interacting with VirusTotal's APIs is through the use of HTTP requests and corresponding JavaScript Object Notation (JSON) object responses (VirusTotal

[2]https://www.virustotal.com/

Team, 2015c). Users of the public API are limited to the four call types listed in Table 2.1, and may only submit four requests per minute. All requests must be accompanied by the user's public API key, as well as additional parameters specific to each type of request. Requests that target a single malware sample must identify it by including either its MD5 or SHA1 hash.

Table 2.1: Summary of VirusTotal's public API calls

API Call	Parameters	Description
`/vtapi/v2/file/scan`	API key, file contents	Upload and scan a given file
`/vtapi/v2/file/rescan`	API key, hash	Rescan a previously submitted file
`/vtapi/v2/file/report`	API key, hash	Get the scan results for a specified file
`/vtapi/v2/comments/put`	API key, hash, comment	Upload a comment for a specified file

2.7.2 Private API

VirusTotal's private API is a billed service that allows commercial clients to circumvent the request rate limitation described in the previous section (VirusTotal Team, 2015b). Users are charged a monthly fee based on the maximum number of requests that they expect to make each month, and can make these requests as often as required. The service's use of Google's scalable App Engine[3] infrastructure guarantees request throughput and ensures availability.

In addition to providing an unlimited request rate, the private API exposes a more comprehensive and powerful set of features. Users with a private key can make use of the extended collection of API calls listed in Table 2.2 (VirusTotal Team, 2015b). Of particular interest to this research are the `search` and `download` calls, which, when used in conjunction, allow a user to retrieve samples that meet a specified criteria. This functionality was leveraged to source RATs written in PHP during the implementation of the download scripts, which are described in Section 3.2.

2.8 Parsing and Lexical Analysis

Parsing is defined as the process of analysing a string of symbols to determine whether it conforms to the rules laid out by a formal grammar (Aho *et al.*, 1986). In the field of Computer Science, the first step in the parsing process is referred to as lexical analysis, which is the process of converting a string of symbols into a sequence of meaningful tokens (Terry, 2005). In PHP, lexical analysis is carried out by the Zend Engine, an open source interpreter originally developed by Andi Gutmans and Zeev Suraski (Ullman, 2004).

[3] https://cloud.google.com/appengine/

Table 2.2: Summary of VirusTotal's private API calls

API Call	Parameters	Description
`/vtapi/v2/file/scan`	API key, file	Upload and scan a given file
`/vtapi/v2/file/rescan`	API key, hash	Rescan a previously submitted file
`/vtapi/v2/file/report`	API key, hash	Get the scan results for a specified file
`/vtapi/v2/file/behaviour`	API key, hash	Get a report about a file's behaviour in a sandbox environment
`/vtapi/v2/file/network`	API key, hash	Get a dump of the network traffic generated by a file when executed.
`/vtapi/v2/file/search`	API key, query, offset	Search for samples that match certain criteria
`/vtapi/v2/file/download`	API key, hash	Download the specified file
`/vtapi/v2/comments/put`	API key, hash, comment	Upload a comment for a specific file
`/vtapi/v2/comments/get`	API key, hash	Get the comments for a specific file

2.8.1 Tokenizer Extension

PHP's Tokenizer extension (PHP Group, 2015h) provides an interface to the lexical analyser used by the Zend Engine[4]. Using this interface, it is possible to carry out token-based source code analysis and modification without the need for a custom parser. Of particular interest to this research are the `token_get_all()` function and the `T_FUNCTION` token type, which can be used in combination to locate and extract function names and bodies. Sections 3.5.2 and 3.5.1 contain more detail as to how this can be achieved.

The `token_get_all()` function can be used to convert a given source string into a stream of PHP language tokens using the Zend engine's lexical scanner (PHP Group, 2015h). These tokens can then be queried using the `token_name()` function to determine their type.

2.8.2 PHP-Parser

PHP-Parser[5] is an open-source parser capable of programmatically manipulating PHP code. Based on the `token_get_all()` function discussed in Section 2.8.1, it is capable of constructing representative abstract syntax trees (ASTs) that encapsulate the structure of the code used as input (Baxter *et al.*, 1998). Each node in an AST represents a syntactic construct within the code. Once constructed, these ASTs can then be transformed back into PHP code according to a predefined set of formatting rules, a process known as pretty printing (Roy and Cordy, 2008).

[4] http://www.zend.com/
[5] https://github.com/nikic/PHP-Parser

2.9 Similarity Analysis

All code dissection techniques can be classified as being either static or dynamic in nature (Binkley, 2007). Static analysis approaches attempt to examine code without running it (Christodorescu *et al.*, 2007). Because of this, these approaches have the benefit of being immune to any potentially malicious side effects. The lack of runtime information such as variable values and execution traces does limit the scope of static approaches, but these are still useful for exposing the structure of code and comparing it with previously analysed samples (Zaremski and Wing, 1993a). By contrast, dynamic approaches extract information about a program's functioning by monitoring it during execution (Christodorescu *et al.*, 2007). These approaches examine how a program behaves and are best confined to a virtual environment such as a sandbox so as to minimise the exposure of the host system to infection (Christodorescu *et al.*, 2007). The remainder of this section introduces several common static and dynamic similarity analysis techniques.

2.9.1 Signature Matching

A software signature is a characteristic byte sequence that can be used to uniquely identify a piece of code (Zaremski and Wing, 1993a). Anti-malware solutions make use of static signatures to detect malicious programs by comparing the signature of an unknown program to a large database containing the signatures of all known malware – if the signatures match, the unknown program is flagged as suspicious. This kind of detection can easily be overcome by making trivial changes to the source code of a piece of malware, thereby modifying its signature (Zaremski and Wing, 1993b).

2.9.2 Pattern Matching

Pattern matching is a generalised form of signature matching in which patterns and heuristics are used in place of signatures to analyse pieces of code (Zaremski and Wing, 1993a). This allows pattern matching systems to recognise and flag code that contains patterns that have been found in previously analysed malware samples, which, although an improvement on signature matching, is still insufficient to identify newly developed malware (Zaremski and Wing, 1993a). Patterns that are too general will lead to false positives (benign code that is incorrectly classified as malicious), whereas patterns that are too specific will suffer from the same restrictions faced by signature matching (Zaremski and Wing, 1993a).

2.9.3 API Hooking

API hooking is a technique used to intercept function calls between an application and an operating system's different APIs (Sun *et al.*, 2006). In the context of code dissection, API

hooking is usually carried out to monitor the behaviour of a potentially malicious program (Berdajs and Bosnic, 2010). This is achieved by altering the code at the start of the function that the program has requested access to before it actually accesses it and redirecting the request to the user's own injected code (Berdajs and Bosnic, 2010). The request can then be examined to determine the exact behaviour exhibited by the program before it is directed back to the original function code (Sun *et al.*, 2006).

The precision and volume of code required for correct API hooking mean that behaviour monitoring systems that make use of the technique are complex and time consuming to implement (Berdajs and Bosnic, 2010). They are also virtually undetectable and thoroughly customisable (only functions relevant to behaviour analysis need be hooked).

2.9.4 Cryptographic versus Approximate Hashing

Hashing is a technique commonly used in forensic analysis that transforms an input string of arbitrary length into a fixed-length signature (Kornblum, 2013). Once generated, these signatures can be used to match identical files efficiently. Traditional cryptographic hashing algorithms such as MD5 and SHA256 are designed such that changing just one bit in the input file leads to the generation of a completely different hash signature. This approach, although ideal for matching identical files, makes these algorithms incapable of matching files that are merely similar. For this purpose, it is necessary to use Context-Triggered Piecewise Hashing (CTPH), an implementation of which is described in the following section.

2.10 The Ssdeep Fuzzy Hashing Algorithm

Ssdeep[6] is a hashing tool that was developed by Kornblum (2006). It is capable of using CTPH to generate fuzzy hashes that can then be compared to determine the similarity of a set of files. The similarity value that the tool generates represents the edit distance between two fuzzy hashes (i.e., the number of changes that need to be made to convert the one hash into the other). As a result of its combination of both rolling and piecewise hashes, the tool's hashing algorithm is more computationally demanding than other algorithms such as MD5, but it is a far more effective way of identifying code reuse in similar files.

2.10.1 Piecewise Hashes

Piecewise hashing is the process of using an arbitrary hash function to generate multiple hashes for a given file instead of only one (Baier and Breitinger, 2011). This is achieved by dividing the

[6] http://ssdeep.sourceforge.net/

file into fixed-size segments and then hashing these segments individually. Originally developed by Nicholas Harbour in 2002, the technique was originally intended to mitigate errors during forensic imaging by confining the effect of a segment error to only one of the piecewise hashes, thereby preserving the integrity of the remaining hashes. Kornblum (2006) recognised the potential of piecewise hashing for constructing fuzzy hashes, and combined it with a rolling hash function to produce the Ssdeep hashing algorithm.

2.10.2 Rolling Hashes

A rolling hash function produces a pseudo-random value based on a contextual window that moves through the input file (Roussev, 2011). During file processing, the value of the hash is thus dependent on the last `n` bytes of the file. As such, a rolling hash can be thought of as representing the "context" of the file at any given time. Algorithms for rolling hashes are constructed such that it is possible to calculate the next hash using only the previous hash, the byte to be removed, and the byte to be added (Chen and Wang, 2008).

2.10.3 Context-Triggered Piecewise Hashing

Traditional piecewise algorithms use fixed offsets to separate a file into blocks which can subsequently be hashed (Baier and Breitinger, 2011). When attempting to compare similar files, however, this approach proves ineffective, as simply adding one byte to the beginning of the input file will shift each of the blocks and alter all of the resulting piecewise hashes. For this reason, the Ssdeep algorithm uses a rolling hash to delineate piecewise hashing blocks (Kornblum, 2006). Each time the rolling hash produces a trigger value, a piecewise hash is recorded and the process beings again. The end result of this process is a CTPH signature in which each value depends on only part of the input. Because of this property, these signatures can be used to detect commonalities between files that are not identical.

2.10.4 Comparing Hashes and the Blocksize Limitation

Hash signatures produced by the Ssdeep algorithm are compared using the Levenshtein or edit distance, which is defined as the minimum number of single-character edits required to change one string of characters into another (Yujian and Bo, 2007). Because the trigger values for the piecewise hash are based on the block size (which is in turn based on the size of the input file), only files with the same block size can be compared in this way (Kornblum, 2006). For this reason, each CTPH signature produced by the Ssdeep algorithm includes hashes based on two block sizes `b` and `2b` so as to facilitate the comparison of signatures containing block sizes within a power of two.

Table 2.3: Pydeep wrapper functions

Function	Description
`pydeep.hash_buf()`	Returns the Ssdeep hash for a given buffer
`pydeep.hash_file()`	Returns the Ssdeep hash for a given file
`pydeep.compare()`	Calculates a similarity value for two hashes

Another documented issue[7] with the current version of the software, Ssdeep 2.5, is the minimum block size that is assigned by the algorithm. During testing carried out for this research, it was discovered that files containing fewer than 200 characters do not register the correct level of similarity. The effect worsens as the file sizes decrease, culminating in a similarity value of zero for identical files containing 18 or fewer characters.

2.10.5 Pydeep

Pydeep[8] is a Python module that includes wrapper functions for the Ssdeep library, which was originally written in C. It allows the user to compute Ssdeep hashes for both files and buffers, and is able to compare two such hashes to determine an overall similarity score. A list of the functions provided by Pydeep is given in Table 2.3.

2.11 Data Visualisation

Data visualisation is the process of representing mundane data (such as numerical values) as visual objects with the aim of increasing accessibility and understanding (Friendly and Denis, 2001). Successful visualisation techniques should assist the viewer with analytical tasks such as making comparisons and identifying patterns in data (Schroeder *et al.*, 2004; Fayyad *et al.*, 2002).

A key concern when considering different visualisation methods is to choose a representation that is well-suited to both the dataset and the desired form of analysis. Each of the many available forms of visualisation lend themselves to a particular type of analysis (Buja *et al.*, 1996). Line graphs, for example, are most suited to displaying data that changes over time, and are useful for discovering overall trends, while bar graphs are generally used to compare different quantities. Bearing this in mind, Sections 2.11.1 through to 2.11.3 introduce three visualisation techniques that are adept at representing similarity between objects. When used in conjunction with one another, these techniques are able to provide a comprehensive overview of sample similarity, and can be used to easily identify meaningful inter-sample relationships within large datasets. Section 2.11.4 details the Python libraries that were used to create these visualisations.

[7] https://github.com/treffynnon/lib_mysqludf_ssdeep/issues/3
[8] https://github.com/kbandla/pydeep

2.11.1 Similarity Matrices

A similarity matrix is a set of similarity scores between pairs of objects from a given object collection (Hartigan, 1967). Each element in the matrix represents the level of similarity between two of the objects, and is calculated using a specified measure of similarity. Similarity matrices are primarily used to identify clusters of similarity in a given set of data, and as such find application in a wide variety of fields, including bioinformatics, where they are used to detect similar Deoxyribonucleic Acid (DNA) sequences, and phylogenetic analysis, where they form the basis for hierarchical clustering techniques (Campanella *et al.*, 2003; Iam-on *et al.*, 2008).

Figure 2.2 displays an example of a rudimentary similarity matrix that contains the pairwise similarity values for four objects A, B, C, and D ranging from 0 to 100. From the diagram it is trivial to determine that the highest level of similarity is observed between samples A and B with a score of 92, while samples B and D display the lowest level of similarity with a score of only 26. The example matrix also demonstrates two properties that are inherent to all pairwise similarity matrices: it is diagonally symmetrical from left to right, and the values along this diagonal all indicate perfect similarity as a result of comparing samples against themselves.

D	47	26	68	100
C	78	40	100	68
B	92	100	40	26
A	100	92	78	47
	A	B	C	D

Figure 2.2: Example of a simple similarity matrix

2.11.2 Heatmaps

A heatmap is a two-dimensional graphical representation of a matrix of values in which each value is represented by a colour or shade (Brodlie *et al.*, 2012). The magnitude of each value can either be denoted by the colour itself (in which case a legend of colours and their associated value ranges needs to be consulted) or by the relative lightness and darkness of the shade, with darker colours traditionally representing larger values. Because of this property, the structures

can be used to easily identify values (or areas of values) that represent a high level of pairwise similarity (Wilkinson and Friendly, 2009).

As is the case with the similarity matrices defined in Section 2.11.1, heatmaps are primarily used in fields that require the comparison of large amounts of data. According to Rajaram and Oono (2010), the clustered heatmap is the most popular technique for visualising genomic data as a result of its ability to display large datasets intuitively and facilitate the detection of structural relationships within the data.

Figure 2.3 displays a simple heatmap created from the similarity matrix shown in Figure 2.2. Although the detail of the original construct is somewhat reduced by substituting the values for shades of blue, the resulting representation is more intuitive and easier to interpret. The increase in accessibility is even more pronounced for larger datasets – complex matrices containing many values are difficult to interpret, but their associated heatmaps can readily be used to identify patterns and relationships. Further proof of this was encountered during the testing performed in Section 4.7, which demonstrated that the more visual representations (i.e., heatmaps and dendrograms) remained useful even when performing similarity analysis among hundreds of RAT samples.

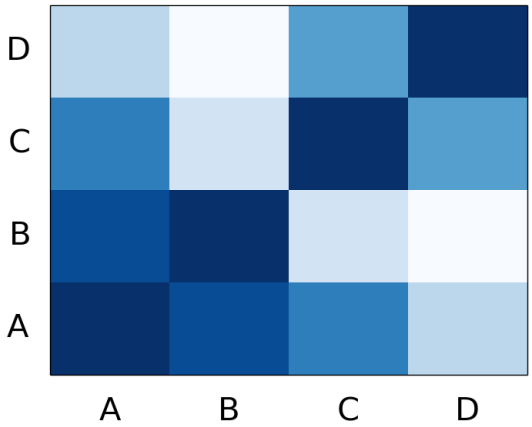


Figure 2.3: Simple heatmap based on the similarity matrix in Figure 2.2

The heatmap in Figure 2.3 can be used to reach the same conclusions that resulted from an examination of the similarity matrix in Figure 2.2. The darkest blocks along the diagonal from left to right demonstrate the perfect similarity between samples when compared to themselves, and the second darkest blocks present at the intersections of samples A and B show that they share the highest level of similarity between unique samples. Samples B and D are the most dissimilar, as is evidenced by the lack of colour at their points of intersection.

2.11.3 Dendrograms

Dendrograms are tree-like structures that can be used to display the relationships that result from hierarchical clustering algorithms in an intuitive way (Everitt and Skrondal, 2002). The hierarchical nature of the dendrograms produced in this way allows for the identification of derivative sample relationships, as well as the magnitude of such relationships (Sokal and Rohlf, 1962).

Figure 2.4a shows a simple dendrogram that is based on the similarity matrix shown in Figure 2.2. An annotated version of the same diagram is shown in Figure 2.4b for explanatory purposes. The horizontal axes of the dendrograms represent the objects and clusters, while the vertical axes represent the distance or dissimilarity between them.

Dendrograms can be used to identify clusters of samples based on their similarity. The two clusters in Figure 2.4b are marked by areas X and Y. Cluster X contains samples A and B, while cluster Y contains samples C and D. Samples in cluster X have more in common with each other than they do with samples in cluster Y. Sample A is thus more similar to sample B than it is to either C or D, and vice versa.

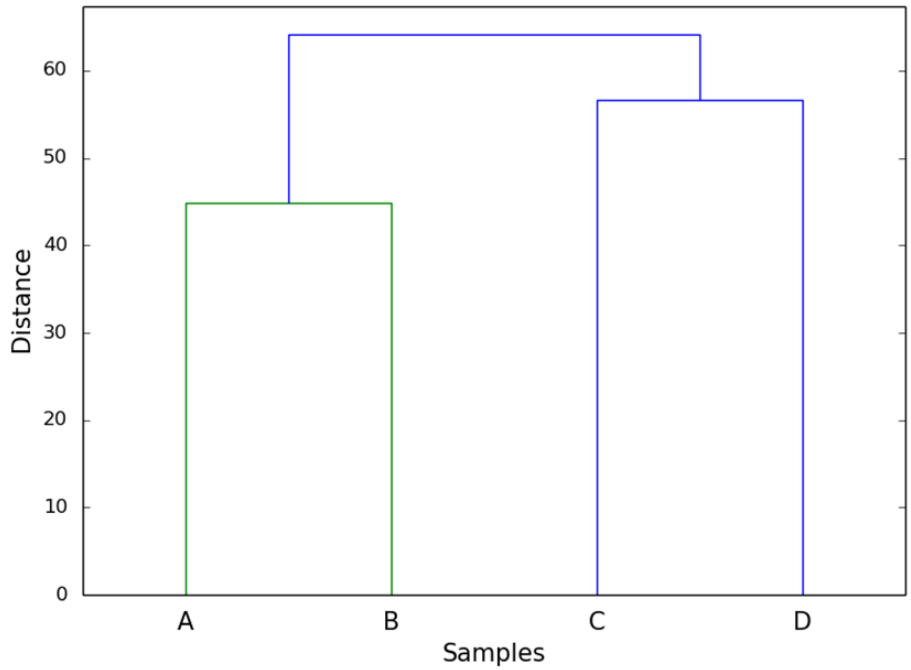
The heights of the branches connecting each cluster represent the distances between them (Choi *et al.*, 2010). The closer the branch is to the x-axis, the lower the distance, and the more similar the samples. A comparison of the heights of clusters X and Y in Figure 2.4b (denoted by lengths E and F, respectively) reveals that A is more similar to B than C is to D.

2.11.4 Data Visualisation Tools

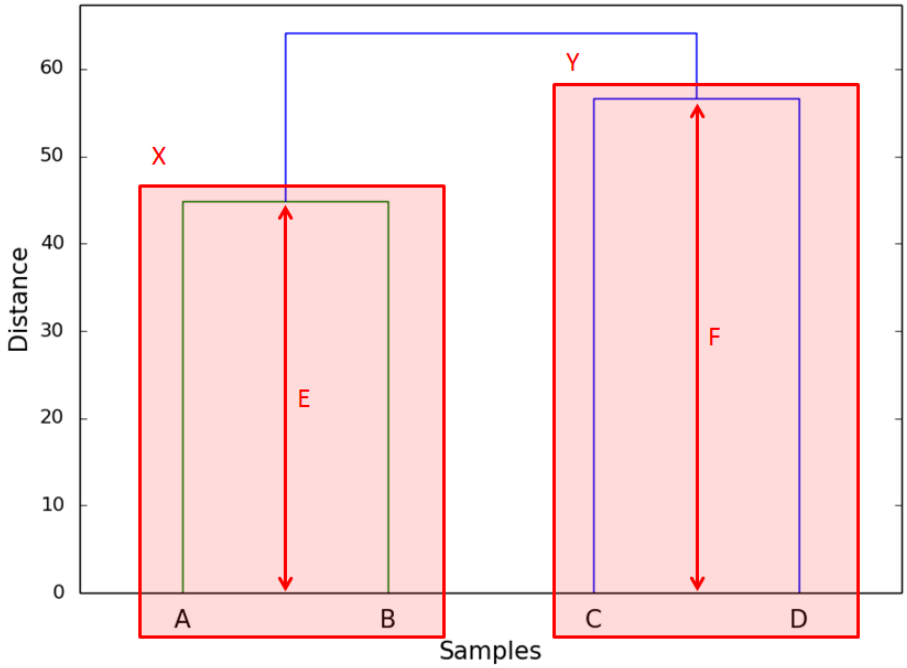
The Python programming language contains a number of extensions that directly support the creation of the data visualisations discussed in Sections 2.11.1 to 2.11.3, including the NumPy[9], SciPy[10], and Matplotlib[11] packages, each of which is briefly described below.

- **NumPy** is Python's fundamental N-dimensional array package. Originally derived from the Numeric extension created by Jim Hugunin in 1995, the package provides support for multi-dimensional arrays and includes a comprehensive set of functions for creating, manipulating, and storing these structures (Oliphant, 2006; Van Der Walt *et al.*, 2011). NumPy forms the base of the SciPy stack, which includes libraries such as Matplotlib, Pandas[12], and SymPy[13] (McKinney, 2012).

[9] http://www.numpy.org/
[10] http://www.scipy.org/
[11] http://matplotlib.org/
[12] http://pandas.pydata.org/
[13] http://www.sympy.org/en/index.html



(a) Original dendrogram based on the similarity matrix in Figure 2.2



(b) Annotated dendrogram based on the similarity matrix in Figure 2.2

Figure 2.4: Original and annotated dendrograms based on the similarity matrix in Figure 2.2

- The **SciPy** library is a collection of a open-source software that is useful in the fields of mathematics, science, and engineering (Bressert, 2012). Created by Travis Oliphant (an active developer of NumPy), the library contains modules that enable tasks that are commonly encountered in these fields, including optimisation algorithms, interpolation, image processing, and hierarchical clustering, amongst others. Scipy's core data structure is the N-dimensional NumPy array, which it uses to store arbitrary data types.
- **Matplotlib** is a cross-platform two-dimensional graphics environment that forms part of the SciPy library and is used to create interactive figures in Python (Hunter, 2007). The environment enables the creation of a variety of visual objects such as bar charts, histograms, heatmaps, scatterplots, and dendrograms, and can be extended to create other plots through the use of external toolkits (Hunter and Dale, 2007).

2.12 Related Work – Deobfuscation and Normalisation

Although the existence and widespread use of code obfuscation is well-documented (Collberg *et al.*, 1997; Christodorescu *et al.*, 2005; Wrench and Irwin, 2015a), relatively few fully-functional deobfuscation tools have been developed to address this problem. Research in this area tends to be largely theoretical, particularly when it comes to the analysis of malware source code. The deobfuscation systems that were found tended to target malware binaries as opposed to their source code. Furthermore, the author was unable to find any deobfuscation systems that explicitly target the PHP language. For this reason, the remainder of this section outlines some of the generic deobfuscation and normalisation tools that are described in the literature.

2.12.1 LOCO: An Interactive Code (De)obfuscation Tool

LOCO is a interactive graphical environment in which a user can experiment and observe the effects of both obfuscation and deobfuscation transformations (Madou *et al.*, 2006). Based on a visualisation tool called Lancet and an obfuscation infrastructure called Diablo, the environment is able to expose the control flow of a program and show the effects of any obfuscating or deobfuscating actions on it. Users can choose either to execute and evaluate existing obfuscation/deobfuscation transformations or to develop and test transformations of their own. The environment's visualisation feature is particularly helpful when it comes to identifying flaws in deobfuscation transformations, as the user can step through the program and identify the effects of the transformation at any point in the code. It also facilitates the manual deobfuscation of programs by allowing users to modify the source code and observe how each modification affects the flow of control.

Although LOCO includes powerful transformation testing and visualisation features, it is more a tool for developing and testing deobfuscation systems than a system in itself. It lacks the

ability to store and reuse code transformations, and its built-in deobfuscation algorithms are designed to be extensible rather than comprehensive. LOCO also functions at the assembly level, which gives it more flexibility but means that its algorithms cannot be adapted for use in deobfuscation systems that function at a higher level.

2.12.2 Deobfuscator: An Automated Control Flow Simplifier

Developed by Raber and Laspe in 2007, Deobfuscator is an IDA Pro[14] plug-in that can be used to simplify malware binaries (Raber and Laspe, 2007). It relies on the extensive debugging capabilities of the IDA Pro disassembler to iteratively simplify instruction sequences by removing unnecessary no-ops and jumps. Once these redundant obfuscation constructs have been removed, the plug-in collapses the remaining instructions to create an optimised (or deobfuscated) set of instructions.

The major advantage of Deobfuscator is that it can be used to deobfuscate binaries written in any language at the assembly level. This is especially useful when the malware's source code is not available, which is often the case. In a system that combines both obfuscation and similarity analysis, however, it is preferable to investigate malware samples at a higher level if possible, as the richer semantic content of the source code provides a more nuanced and detailed basis for comparison.

2.12.3 A Malware Transformer to Improve Detection Rates

In an effort to improve the detection rates of commercial malware detectors, Christodorescu *et al.* (2007) designed and implemented a malware transformer that can be used to reverse common obfuscation techniques at the assembly level. As was the case with the Deobfuscator tool described in the previous section, the transformer relies on disassembly processes carried out by IDA Pro. In addition to the removal of single-instruction no-ops and jumps, however, the transformer is also able to perform three more complex deobfuscation tasks, including code reordering, code extraction, and the removal of semantic no-ops.

Although more capable than Deobfuscator, the malware transformer developed by Christodorescu *et al.* (2007) also operates at the assembly level and is thus of limited value as a precursor to source code similarity analysis. The focus of the tool is to improve malware detection rates for various antivirus solutions, and not to expose additional code for the purposes of comparison, which is one of the primary goals of this research.

[14] https://www.hex-rays.com/products/ida/

2.12.4 Summary

All of the deobfuscation systems reviewed in this section were designed to improve the results achieved by commercial antivirus engines (with the notable exception of LOCO, which is used to test the deobfuscation systems themselves). Because most malware is distributed in binary form, these systems rely on deobfuscation at the assembly level. Operating at this level is undeniably advantageous in a production environment, but the generic nature of these tools means that they are unable to reverse language-specific obfuscation constructs such as those described in Section 2.6.

2.13 Related Work – Malware Similarity Analysis

The overwhelming increase in the number of malicious software variants in recent years has spurred research into automated detection methods, most of which incorporate some form of heuristic or similarity analysis techniques (Baxter *et al.*, 1998; Bailey *et al.*, 2007; Sharif *et al.*, 2008a; Lu and Debray, 2012; Wrench and Irwin, 2015a). As was the case when examining related work on code deobfuscation in Section 2.12, none of these approaches target PHP specifically, but a multitude of generic malware analysis tools have been developed and tested (Walenstein *et al.*, 2007; Shankarapani *et al.*, 2011; Nair *et al.*, 2010; Jang *et al.*, 2011). An overview of four of the tools that were found to be most relevant to this research are briefly outlined in the remainder of this section.

2.13.1 BitShred: Scalable Malware Analysis using Feature Hashing

Developed by Jang *et al.* (2011) in response to the growing number of malware variants faced by modern security vendors, BitShred is a scalable analysis framework that uses feature hashing to expedite the evaluation of large numbers of malware samples. The system is able to compare binaries by performing either n-gram (i.e., string sequence) or behavioural analysis and then reducing the resulting feature space by hashing each of the uncovered features. These hashed features can then be compared to those of other samples more efficiently than the high dimensional feature spaces that are traditionally used in malware analysis. The system incorporates work undertaken by Abou-Assaleh *et al.* (2004) and Bayer *et al.* (2009) in the fields of n-gram and behavioural analysis, respectively.

The main goal of the BitShred analysis framework is to increase the scalability of existing malware analysis techniques by using feature hashing to improve the efficiency of all inter-sample comparisons. As such, it is more an enabler of similarity analysis for large datasets than a complete similarity analysis tool. Despite this, tests of the system using n-gram and behavioural

analysis showed that the tool more than doubled the efficiency of these two approaches while maintaining the same levels of accuracy as the original research.

2.13.2 Vilo: Code Reuse Detection using N-gram and N-perm Analysis

Walenstein *et al.* (2007) used code reuse detection techniques on disassembled code in an attempt to detect new malware variants. The result of their efforts was Vilo, a method for detecting code reuse based on the comparison of both n-grams (ordered string sequences) and more general n-perms, where the order of the strings is not considered when performing matching operations. By combining this approach with feature weighting algorithms that assign more weight to n-grams and n-perms that appear less frequently in the target file, the authors of Vilo were able to achieve a detection rate of 79%, albeit for a fairly limited dataset. As was the case with many of the deobfuscation tools discussed in Section 2.12, this analysis was conducted at the assembly level using the IDA Pro disassembly tool.

2.13.3 SAVE and MEDiC: Malware Detection using API and Assembly Calls

Shankarapani *et al.* (2011) developed two malware detection tools that rely on similarity analysis as opposed to traditional signature-based matching techniques. SAVE, the Static Analyser for Vicious Executables, analyses the sequence of API calls made by a potentially malicious sample and compares it to a database containing the API call sequences of known malware. The Malware Examiner using Disassembled Code (MEDiC) performs a similar comparative analysis using sequences of assembly calls, with the goal of determining the likelihood that the given sample contains malicious code. Although false positives were observed when using both techniques individually, the authors assert that the combination of the two approaches minimizes such errors.

Both the SAVE and MEDiC tools use a single measure of similarity to determine whether a sample contains code sequences corresponding to those discovered in known malware samples (Shankarapani *et al.*, 2011). Files that are processed by the tools are simply classified as being malicious or benign according to a threshold value – no further investigation of inter-sample relationships is carried out. In addition to this, the majority of the testing conducted by the authors was aimed at pitting their tools against existing detection solutions to determine their relative effectiveness rather than at more in-depth similarity analysis.

2.13.4 Medusa: Dynamic Malware Analysis using API Signatures

Medusa is a malware analysis system developed by Nair *et al.* (2010). Like the SAVE tool described in the previous section, the Medusa system traces the API calls made by input samples

to detect malicious behaviour. Instead of comparing these call sequences to those of known malware samples, however, Medusa creates an API call signature based on the combined call sequences of an entire malware class. The system's detection engine then uses these composite class signatures to identify and classify new malware samples.

As with all the other similarity analysis tools discussed in this section, the focus of the Medusa system is the detection of new malware instances based on their similarity to a body of previously classified malicious samples. The goal of such systems is classification rather than interpretation, which means that interrogation of any detected similarity is limited to a predefined threshold value.

2.13.5 Summary

The approaches to similarity analysis presented in this section are all responses to the rapid increase in the proliferation of malware samples in recent years (Kaspersky, 2011; Edem *et al.*, 2014). All focus on providing an efficient means of detecting malicious software based on sample characteristics and behaviour. This emphasis on efficiency and automation means that the similarity process is restricted, with most approaches implementing a single measure of similarity that is designed to scale well with larger datasets. Furthermore, the interpretation of the results was often limited to the classification of each input sample as either malicious or benign, with no room for a gradual scale of similarity. All of these traits, combined with the generic nature of all of the tools that were discussed, make these approaches ideal for use as production antivirus solutions, but they lack the capacity for the more nuanced and detailed analysis that is preferable when attempting to identify groups of derivative malware samples.

2.14 Chapter Summary

This chapter began with a discussion of the noteworthy features of PHP, including its performance and security characteristics as well as its widespread use as a server-side scripting language (and consequent popularity among the developers of RATs). The basic structure and capabilities of a typical web shell were also discussed, with particular emphasis on the delivery vectors that are commonly used to upload these shells onto vulnerable hosts. Various methods of obfuscating code were then presented, as were techniques for reverse engineering scripts obfuscated using these methods. The concept of similarity analysis was introduced, along with a discussion of the different approaches used to perform it effectively. Techniques for using CTPH as a method of similarity analysis were then described before moving on to describe three useful ways of visualising levels of similarity between a large number of files. The chapter concluded with a discussion of related work undertaken by researchers in the fields of code deobfuscation and similarity analysis.

3

Design and Implementation

The detection of derivative PHP-based malware samples required the creation of a system capable of deobfuscating and normalising sample inputs, extracting relevant features for analysis, performing comparisons between sample features, and finally visualising the results. This chapter begins in Section 3.1 by providing a high-level overview of the constructed system, including an explanation of how malware samples pass through it. Section 3.2 details the implementation of the download scripts responsible for retrieving and storing such samples, before Section 3.3 goes on to describe the structure of the decoder component that is used to deobfuscate and normalise these samples prior to similarity analysis.

The Viper malware analysis framework that was used as a basis for the construction of the similarity analysis system is introduced in Section 3.4. Sections 3.5 and 3.6 describe the individual and batch modules that are used to extract pertinent features from each of the malware samples, while Section 3.7 introduces the matrix module that compares these features and produces representative similarity matrices. Section 3.8 concludes the chapter with a description of the visualisation modules, which are used to create graphical representations of the results produced by the matrix module to facilitate the identification of meaningful inter-sample relationships.

3.1 System Structure

The system for detecting derivative malware samples was developed around Viper[1], a unified framework designed to facilitate the static analysis of arbitrary files (Guarnieri, 2015c). This framework was extended through the creation of custom modules: separate scripts that are dynamically loaded each time Viper is launched and which encapsulate a distinct sample processing or analytical capability. Groups of these modules were used to extract relevant features for analysis, compare these features with those of other samples, and finally create and visualise similarity matrices in an attempt to identify relationships within a collection of malware samples. A full list of the modules that were developed for the system is given in Appendix B.

Figure 3.1 demonstrates the path of malware samples through the system. To begin with, samples obtained from the malware collection maintained by the VirusTotal online analysis framework and other Internet sources are passed to the decoder, a component designed to perform static code deobfuscation and normalisation prior to analysis. This component produces raw and decoded versions of each malware sample, both of which are then stored in Viper's central repository. These samples are subsequently processed by the batch modules, which prepare samples for similarity analysis by extracting relevant and comparable features. The batch modules in turn rely on the logic implemented in the individual modules, the latter of which were designed to be run against a single sample at a time.

Once code normalisation and feature extraction have been completed, the matrix module is used to compare these features by creating similarity matrices that represent the observed similarity between a given collection of malware samples. These matrices are passed to the visualisation modules, which generate graphical interpretations of the matrices to assist in the identification of meaningful sample relationships.

A Note on System Design

Although primarily used as a research tool, the similarity analysis system was developed with the ultimate goal of being integrated into the Viper framework as a PHP-specific extension. For this reason, the design choices illustrated throughout this chapter were made based on guidelines laid out by the Viper community. Emphasis was placed on user-friendliness and portability, and features such as usage instructions, exception handling, input validation, and relative file paths were therefore implemented wherever necessary. Although these features are tangential to the core functioning of the system and are not discussed in great detail for the remainder of this chapter, they are noted here for the sake of completeness.

[1] https://github.com/viper-framework/viper

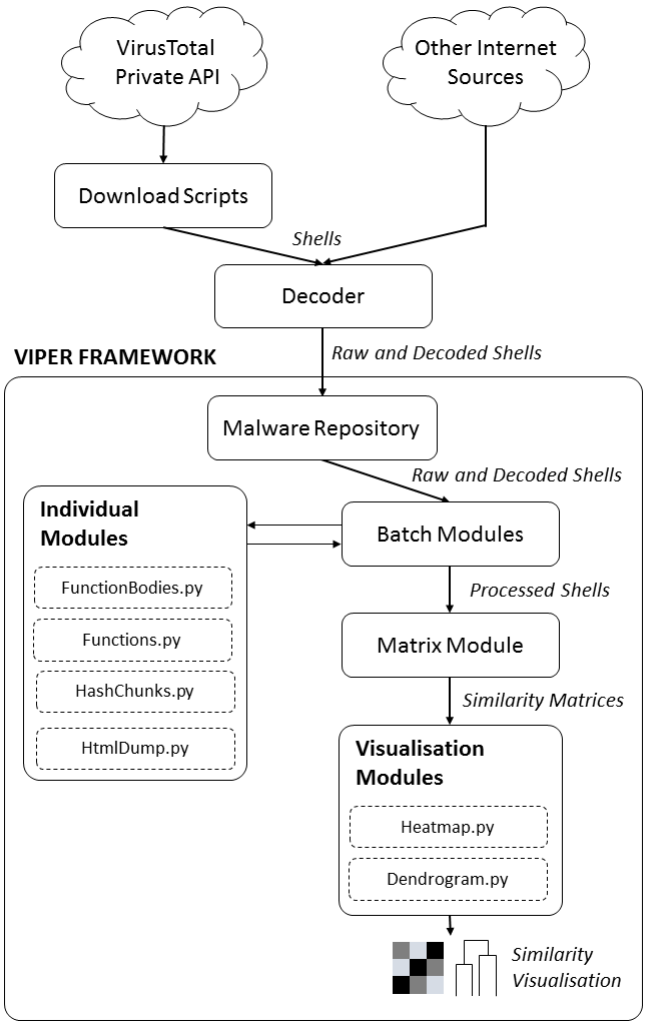


Figure 3.1: System structure

3.2 Download Scripts

Although the malware samples used in this research were obtained from a variety of online sources (see Section 4.1 for a detailed source breakdown), the vast majority were retrieved from the collection of samples maintained by the VirusTotal online analysis service. As discussed in Section 2.7, VirusTotal allows researchers and commercial clients with access to a private API key to download samples that have been submitted by other users. This is achieved by making scripted search and download queries to the service's online API VirusTotal Team (2015b).

Two python scripts were created to automate the sample retrieval process. The first is responsible for creating, encoding, and submitting the search query and adding the results to the local list of hash identifiers, while the second handles the submission of download requests and the addition of the returned samples to the test collection.

3.2.1 Fetch

The purpose of the Fetch.py script is to craft and submit search queries to VirusTotal's private API, and to merge the list of returned hash identifiers with a list of those already in the collection. The query string is parameterised in such a way as to limit the results to RATs written in PHP. Additionally, only samples that have been identified as being malicious by at least one antivirus engine are included in the request. The full parameterised fetch query is shown in Listing 3.1.

```
params = urllib.urlencode({'apikey': key,
  'query': 'type:php engines:"Backdoor:PHP" positives:1+',
  'offset': offset})
```

Listing 3.1: Parameterised VirusTotal fetch query

The `'offset'` parameter in Listing 3.1 is used to paginate the results of the search in cases where more than 300 matches are found. Thus, to download the complete set of 1 969 hashes matching this particular query, it was necessary to store the offset value between query submissions. To avoid the addition of duplicate hashes when the script is run multiple times, this offset is also written to file after every query submission. Listing 3.2 shows how this is achieved in the main loop of the Fetch.py script.

3.2.2 Download

The purpose of the Download.py script is to create separate retrieval requests for each hash in the current list of hashes, and to store the results of these requests as they are processed. Each download request requires the MD5 hash of the desired sample, as is shown in Listing 3.3.

Unlike the search queries described in Section 3.2.1, download requests must be made individually and each returns the binary content of a single file. This content is then written to file by the script, as is demonstrated by the extract from the Download.py script shown in listing 3.4.

3.3 Decoder

The first of the major components developed for the system was the decoder, which is responsible for performing code deobfuscation and normalisation prior to analysis. Deobfuscation is the process of revealing code that has been deliberately disguised, while code normalisation is the process of altering the format of a script to promote readability and uniformity (Preda and Giacobazzi, 2005). Figure

```
...

while(True):
  # Specify the query parameters
    params = urllib.urlencode({'apikey': key,
      'query': 'type:php engines:"Backdoor:PHP" positives:1+',
      'offset': offset})

    # Submit the query
    try:
        request = urllib2.Request(url, params)
        response = urllib2.urlopen(request)
        response_data = response.read()
    except URLError as e:
        print("Failed: {0}".format(e))
        break

    # Extract the JSON object
    try:
        data = json.loads(response_data)
    except ValueError as e:
        print("Failed: {0}".format(e))
        break

    # If successful, write the hashes and the offset to file
    if data['response_code'] == 1:
        with open(hash_file, 'a') as f:
            for item in data['hashes']:
                f.write(item + '\n')
        offset = data['offset']
        with open(offset_file, 'w') as f:
            f.write(offset)
    else:
        break

...
```

Listing 3.2: Fetch.py extract demonstrating the hash retrieval loop

The decoder is considered a static deobfuscator in that it manipulates the code without ever executing it. The advantage of this approach is that it suffers from none of the risks associated with malicious software execution, such as the unintentional inclusion of remote files, the overwriting of system files, or the loss of confidential information. Static analysers are however unable to access runtime information (such as the value of a variable at any given time or the current program state) and are thus limited in terms of behavioural analysis.

The purpose of this component is to expose the underlying program logic of a given shell by removing any layers of obfuscation that may have been added by the shell's developer. This process is controlled by the decode function, which is described in Section 3.3.1. It makes use of two core supporting functions, `processEvals()` and `processPregReplace()`, the details of which are provided in Sections 3.3.2 and 3.3.3, respectively. Post-processing correction and nor-

```
  params = urllib.urlencode({'apikey': key, 'hash': line.strip() })
```

Listing 3.3: Parameterised VirusTotal download query

```
1  ...
2
3  for line in hashes:
4    # Specify the query parameters
5      params = urllib.urlencode({'apikey': key, 'hash': line.strip() })
6
7      # Submit the query
8      try:
9          request = urllib2.Request(url, params)
10         response = urllib2.urlopen(request)
11         response_data = response.read()
12     except URLException as e:
13         print("Failed: {0}".format(e))
14         continue
15
16     # Store the downloaded sample
17     with open(download_folder + line, 'w') as f:
18         f.write(response_data)
19
20 ...
```

Listing 3.4: Download.py extract showing the file retrieval and storage loop

malisation of code is performed by the `normalise()` function described in Section 3.3.4, while the statistics gathered during the deobfuscation process are recorded by the `WriteStats()` function, which is described in Section 3.3.5.

The class diagram for the decoder component is shown in in Figure 3.2. In addition to the five previously described functions, the component also keeps track of seven main global variables, including `$original`, which stores the raw version of the input file, and `$decoded`, which contains the deobfuscated and normalised version after processing. The five remaining variables are used to record statistics relating to the deobfuscation process, including the time taken to process the file, the depth of the obfuscation, the number of `eval()` and `preg_replace()` constructs that were encountered, and the different combinations of auxiliary string manipulation functions that were used in within the `eval()` functions.

3.3.1 decode()

The part of the decoder responsible for removing layers of obfuscation from PHP shells is the `decode()` function. It scans the code for the two functions most associated with obfuscation, namely `eval()` and `preg_replace()`, both of which are capable of arbitrarily executing

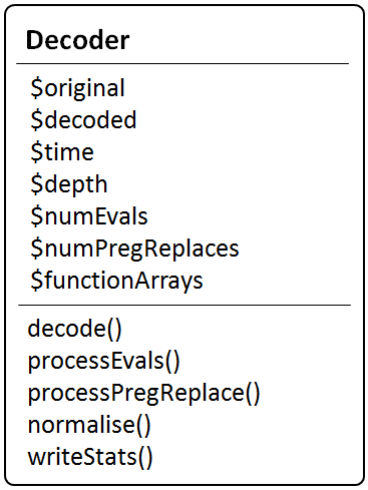


Figure 3.2: Class diagram for the decoder component

PHP code. The `eval()` function interprets its string argument as PHP code, while `preg_replace()` can be made to perform an `eval()` on the result of its regular expression search and replace by including the deprecated `'/e'` modifier. Furthermore, `eval()` is often used in conjunction with auxiliary string manipulation and compression functions in an attempt to further obfuscate the code (see Section 3.3.2 for more details on how this is achieved).

Once an `eval()` or `preg_replace()` is found in the script, either the `processEvals()` or the `processPregReplace()` helper function is called to extract the offending construct and replace it with the code that it represents. To deal with nested obfuscation techniques, this process is repeated until neither of the functions is detected in the code. Some code normalisation is then performed to transform the output into a readable format before the decoded shell is stored in the database alongside its raw counterpart. The complete pseudo-code of this process is presented in Listing 3.5.

```
BEGIN
  Format the code
  WHILE there is still an eval or preg_replace
    Increment the obfuscation depth
    Process the eval(s)
    Format the code
    Process the preg_replace(s)
    Format the code
  END WHILE

  Perform normalisation
  Store the decoded shell in the database
END
```

Listing 3.5: Psuedo-code for the `decode()` function

Both the `processEvals()` and `processPregReplace()` functions rely on string processing techniques to locate and extract `eval()` and `preg_replace()` constructs. Extraneous whitespace characters can cause these techniques to malfunction and extract sections of code that are not related to a given construct, and must thus be removed from the input script prior to deobfuscation. This process is repeated after each iteration of the `processEvals()` and `processPregReplace()` functions to ensure that whitespace is also removed from the newly-visible sections of code.

3.3.2 processEvals()

The `eval()` function is able to evaluate an arbitrary string as PHP code, and as such, is widely used as a method for obfuscating code. The function is so commonly exploited that the PHP group includes a warning against its use – it is recommended that it only be used in controlled situations, and that user-supplied data be strictly validated before being passed to the function. (PHP Group, 2015b)

As described in Section 2.6, authors of malicious software often use the `eval()` function in conjunction with other string manipulation functions to further frustrate reverse engineering attempts. These functions typically compress, encode, or otherwise modify the string argument to increase the complexity of the obfuscation, thereby increasing its resilience to automated analysis. The `processEvals()` function is able to detect and perform some of the more common string manipulation functions in an attempt to reveal the obfuscated code. The functions that `processEvals()` is currently able to detect and process are listed in Table 3.1.

Table 3.1: Auxiliary string manipulation functions handled by `processEvals()`

Auxiliary Function	Description
`base64_decode()`	Decodes data encoded using `base64_encode()`
`gzinflate()`	Inflates a string compressed using `gzdeflate()`
`gzuncompress()`	Decompresses data packed using `gzcompress()`
`str_rot13()`	Restores a string encoded using `str_rot13()`
`strrev()`	Restores a string reversed using `strrev()`
`rawurldecode()`	Decodes a string encoded using `rawurlencode()`
`stripslashes()`	Unescapes a string
`trim()`	Strips whitespace from the beginning and end of a string

The `processEvals()` function was designed to be extensible. At its core is a switch statement used to apply auxiliary functions to the string argument. Adding another function to the list already supported by the system can be achieved by simply adding a case for that function. In future, the system could be extended to apply functions that it has not encountered before or been explicitly programmed to deal with. This possible extension to the system is discussed in more detail in Chapter 4.

Listing 3.6 gives the complete pseudo-code for the `processEvals()` function. To begin with, string processing techniques are used to detect the `eval()` constructs and any auxiliary string manipulation functions contained within them. The `eval()` is then removed from the script and its argument is stored as a string variable. Auxiliary functions are detected and stored in an array, which is then reversed allowing each function to be applied to the argument. The result of this process is re-inserted into the shell in place of the original construct.

```
BEGIN
  WHILE there is still an eval in the script
    IF the eval contains a string argument
      Find the starting position
      Find the end position
      Remove the eval from the script
      Extract the string argument
      Count the number of auxiliary functions
      Populate the array of functions
      Reverse the array

      FOR every function in the reversed array
        Apply the function to the argument
      END FOR
    END IF
    Insert the resulting code
END
```

Listing 3.6: Psuedo-code for the `processEvals()` function

As a static deobfuscation component, the `processPregReplace()` function is unable to process **eval()** constructs with variable arguments, as the value of the argument can only be resolved at runtime. For this reason, the function currently ignores `eval()` statements that contain variables. In future, this limitation could be overcome by providing the `processEvals()` function with runtime information obtained through dynamic analysis, as is discussed in Section 5.3.

3.3.3 processPregReplace()

The `preg_replace()` function is used to perform a regular expression search and replace in PHP (PHP Group, 2015g). The danger of executing the function lies in the use of the `'/e'` modifier. If this modifier is included at the end of the search pattern, the interpreter performs the replacement and then evaluates the result as PHP code, but the system prevents this from happening, as demonstrated in Listing 3.7. Although the `'/e'` modifier was deprecated in version 5.5.0 and will be completely removed in version 7, many of the malware samples in the collection still make use of it.

Listing 3.7 shows the complete pseudo-code of the `processPregReplace()` function, which is tasked with detecting `preg_replace()` calls in a script and replacing them with the code

```
BEGIN
  WHILE there is still a preg_replace
    IF the preg_replace contains string arguments
      Find the starting position
      Find the end position
      Remove the preg_replace from the script
      Extract the string arguments
      Remove the '/e' from first argument
      to prevent evaluation
      Perform the preg_replace
      Insert the deobfuscated code
    END IF
  END WHILE
END
```

Listing 3.7: Psuedo-code for the `processPregReplace()` function

that they were attempting to obfuscate. In much the same way as the `processEvals()` function, string processing techniques are used to extract the `preg_replace()` construct from the script. Its three string arguments are stored in separate string variables and, if detected, the `'/e'` modifier is removed from the first argument to prevent the resulting text from being interpreted as PHP code. The `preg_replace()` can then be safely executed and its result can be inserted back into the script.

3.3.4 normalise()

Many of the outputs of the feature extraction modules described later in this chapter are affected by the layout of the scripts that are passed to them. Furthermore, it was found that the deobfuscation operations performed by the `processEvals()` and `processPregReplace()` functions often produced unpredictable and irregularly formatted code. To mitigate the effects of arbitrary formatting constructs on the results of the similarity analysis process, the `normalise()` function was created.

The purpose of the `normalise()` function is to apply a uniform formatting convention to every shell sample after the deobfuscation process has been completed. An expedient way of achieving this is to pass the script to a PHP parser which creates an AST (as described in Section 2.8.2). All the original formatting is lost during the parsing process, as the AST only stores the lexical tokens found in the script. These tokens can be output according to a predefined set of formatting rules, ensuring that every sample conforms to the same formatting scheme.

Although the Zend engine for interpreting PHP can be used to split source code into a stream of PHP tokens (see Section 2.8.1 for more details about the Tokenizer extension), it lacks the functionality to construct an AST and output it in a uniform way. For this reason, an open source lexical parser and pretty printer called PHP-Parser was used to construct the AST and overwrite

the existing sample text. Listing 3.8 gives the implementation of the layout normalisation function.

```
1  // Initialise parser and pretty printer
2  $parser = new PhpParser\Parser(new PhpParser\Lexer\Emulative);
3  $prettyPrinter = new PhpParser\PrettyPrinter\Standard;
4
5  ...
6
7  function normalise()
8  {
9    global $decoded, $parser, $prettyPrinter;
10
11   // Parse
12   $stmts = $parser->parse($decoded);
13
14   // Pretty print
15   $decoded = $prettyPrinter->prettyPrint($stmts);
16 }
17
18 ...
19
```

Listing 3.8: Decode.php extract showing the implementation of the `normalise()` function

After its instantiation, the lexical parser is used to parse the deobfuscated code into an AST, which is stored in the `$stmts` variable in line 8. The choice of the emulative parser allows the decoder component to parse code from versions of PHP that are newer than the version that it itself is running on. The resulting AST is reformatted by the pretty printer, and the original script is overwritten in line 15.

3.3.5 writeStats()

The purpose of the `writeStats()` function is to record information about the deobfuscation and normalisation processes for future analysis. While processing each file, the decoder keeps track of metrics such as the number of `eval()` and `preg_replace()` constructs it has encountered, as well as the depth of the obfuscation resulting from their use. The time taken to process each sample is also recorded, as is a list of any auxiliary functions used as part of a compound `eval()` statement. The latter is encoded as a JSON object and stored in a separate file for ease of access when calculating cross-sample statistics (refer to Section 4.3 for a detailed description of how this is carried out).

Listing 3.9 details the simple implementation of the `writeStats()` function. The global variables in line 3 are updated throughout the decoding process, and therefore represent the final values that must be written to file. These metrics are later used to determine both the prevalence

of idiomatic obfuscation constructs in the sample collection, as well as the performance of the decoder component as a whole.

```
1  function write_stats()
2  {
3    global $path, $depth, $time, $numEvals, $numPregReplaces, $functionArrays;
4
5    $file = fopen($path."(stats)", "w");
6    fwrite($file, $depth."\n");
7    fwrite($file, $time."\n");
8    fwrite($file, $numEvals."\n");
9    fwrite($file, $numPregReplaces);
10   fclose($file);
11
12   $file = fopen($path."(function_arrays)", "w");
13   $json = json_encode($functionArrays);
14   fwrite($file, $json);
15   fclose($file);
16 }
17
```

Listing 3.9: Decode.php extract showing the implementation of the `writeStats()` function

3.4 Viper Framework

Viper (Guarnieri, 2015c) is a unified framework designed to facilitate the static analysis of arbitrary files. It consists of commands (core functions used to open, close, delete, and tag file samples) and modules, which are dynamically loaded and can be run against either an open file or any number of files from the database. This modular design makes the framework highly extensible – extra functionality can be added by simply creating a new module. It is this extensibility that prompted Viper's use as the basis for this research.

3.4.1 Projects

Malware samples in Viper can be organised into separate projects (Guarnieri, 2015d). Every project maintains its own repository of binary files, and an arbitrary number of projects can be created. All commands and modules in Viper can only be run against samples that form part of the project currently open.

Viper projects are particularly useful when dealing with large malware collections, as they allow specific families of samples to be stored and analysed separately. Once it has been determined that a group of samples share a common feature, it is a simple matter to transfer these samples into a new project for further analysis. Tests run against a smaller selection of samples are more

expedient, and the resulting graphs are more concise, allowing for faster and more accurate conclusions to be drawn.

Projects in Viper can be specified at startup using the `'-p'` argument followed by the name of the project. If the project exists, Viper opens it, otherwise a new project is created. Figure 3.3 demonstrates how an existing project named `'c99'` is opened in Viper.

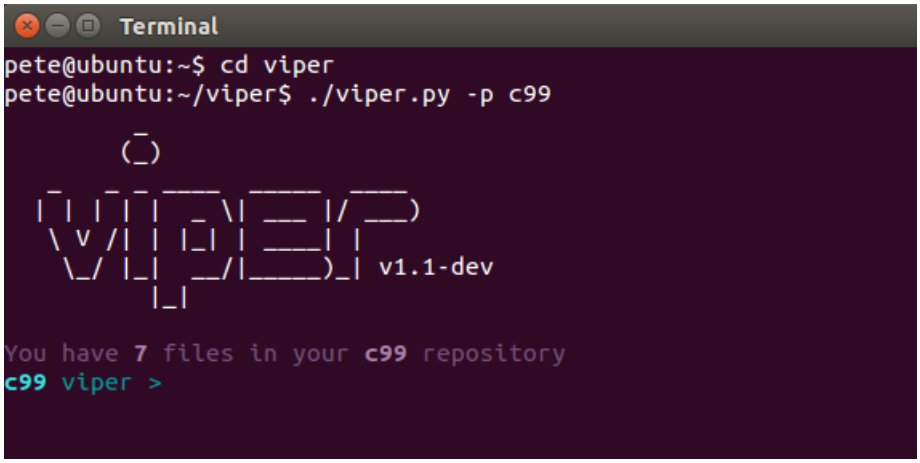


Figure 3.3: Opening an existing project using Viper

It is also possible to list all available projects and switch between them without restarting Viper. The `'projects --list'` command prints a list of Viper projects, while the `'projects --switch'` command switches the project context, as is shown in Figure 3.4.

3.4.2 Sessions

Access to a specific malware sample in Viper is achieved by opening a Viper session (Guarnieri, 2015e), either by searching for the sample by name or by specifying its MD5 hash. Most of the commands and the modules provided in the core Viper framework are designed to be run on a single file and require a session, but any module can access multiple files by retrieving them from the database (see Section 3.4.3 for information on how this is achieved).

Session objects are used to provide modules with information about the sample that is currently open. A global `__sessions__` object provides access to the current session object (`__sessions__.current`), a list of all open session objects (`__sessions__.sessions`), and a list containing the results of the last find command that was executed (see Section 3.4.4 for more information on commands in Viper). A summary of the information that each session object encapsulates is provided in Table 3.2.

Sessions opened during the current execution of Viper can be viewed and switched to in much the same way as the projects described in Section 3.4.1. This functionality is accessed through

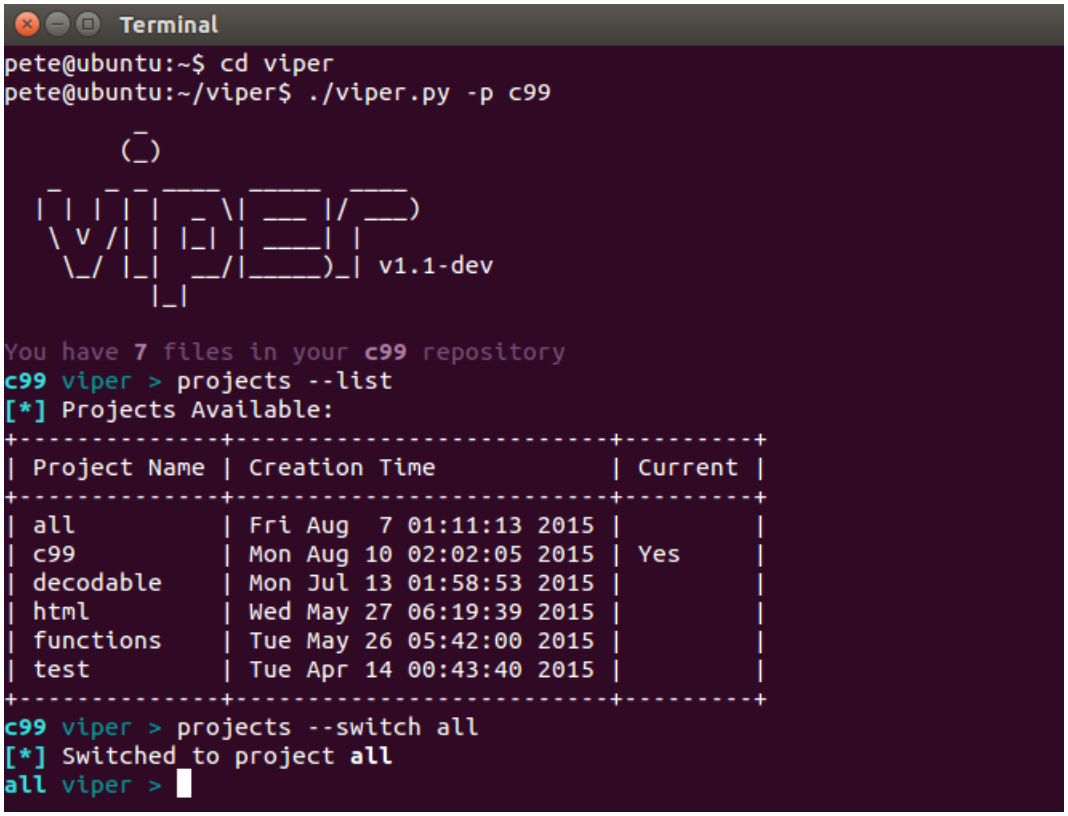


Figure 3.4: Listing and switching between different projects using Viper

the 'sessions --list' and 'sessions --switch' commands, respectively, as is demonstrated in Figure 3.5. Duplicate sessions are not permitted – each time a new session is created using a file that is already open in another session, the older session will be deleted.

The individual modules developed for this research (and described in Section 3.5) all require that an active session be open on the sample that needs to be processed. This is because these modules rely on the session attributes listed in Table 3.2 in order to perform their respective tasks. An extract from the Decode.py module shown in Listing 3.10 demonstrates how the `is_set()` function of the global `__sessions__` object is used to check for the presence of an open session in line 8.

3.4.3 Database

The Viper sessions discussed in the previous section provide a more accessible way to access information about a single sample without resorting to database queries. If a module requires access to multiple samples, it must import and interact with the Database class, which acts as a wrapper for the SQLite[2] database used to organise and store malware samples. Once imported,

[2]https://www.sqlite.org/

Table 3.2: Attributes of a `__session__` object in Viper

Session Attribute	Description
__sessions__.current.file.path	Aabsolute path of the current file
__sessions__.current.file.name	Name of the current file
__sessions__.current.file.size	Size (in bytes) of the current file
__sessions__.current.file.type	Type and encoding of the current file
__sessions__.current.file.mime	MIME type and subtype of the current file
__sessions__.current.file.md5	MD5 hash of the current file
__sessions__.current.file.sha1	SHA-1 hash of the current file
__sessions__.current.file.sha256	SHA-256 hash of the current file
__sessions__.current.file.sha512	SHA-512 hash of the current file
__sessions__.current.file.crc32	CRC-32 check value for the current file
__sessions__.current.file.tags	List of user-defined tags attached to the current file

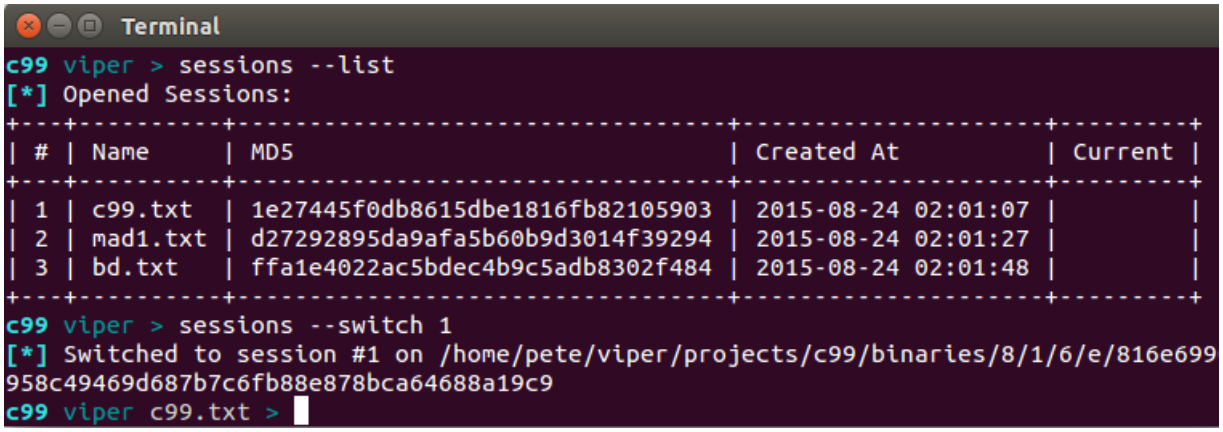


Figure 3.5: Listing and switching between different sessions using Viper

the Database object can be used to access the local project repository through the use of the `find()` function, which accepts a key and a value as search parameters.

The batch modules described in Section 3.6 all make use of the Database class' `find()` function to retrieve and process all samples in a given project. An extract from the Decode_All.py module shown in Listing 3.11 details how this is achieved in lines 13 and 14.

3.4.4 Commands

Simple sample access and modification operations in Viper are carried out using commands (Guarnieri, 2015b). This set of core operations allows a user to open, close, delete, store, or tag an open binary file, as well as display an overview of its characteristics. Table 3.3 details all the available Viper commands and their respective uses.

```
1  class Decode(Module):
2      cmd = 'decode'
3      description = 'Reveals code hidden by eval() or preg_replace() constructs'
4
5      def run(self):
6
7          # Check for an open session
8          if not __sessions__.is_set():
9              print_error('No session opened')
10             return
11     ...
12
```

Listing 3.10: Decode.py extract demonstrating the use of the `is_set()` function

Table 3.3: Viper's core commands

Command	Description
clear	Clears the console window
close	Closes the current session
delete	Deletes the current file
exit	Terminates the current execution of Viper
export	Saves the surrent session to a specified file
find	Searches for a file using a name or hash
help	Displays the help dialogue
info	Display an overview of the current file
new	Creates a new file
notes	Allows notes on the current file to be viewed, edited, or deleted
open	Opens a specified file using either its SHA-1 or MD5 hash
projects	Lists all existing projects
sessions	Lists all open sessions
store	stores a specified file or folder in the local repository
tags	Allows the tags associated with the current file to be viewed, edited, or deleted

3.4.5 Modules

Viper modules are dynamic plugins that encapsulate a distinct analytical capability. Each time the framework is launched, Python files in the modules directory are registered as Viper modules and associated with a specified command to allow them to be accessed through Viper's shell interface. Modules can either avail themselves of the individual shell information contained in an open session object (as explained in Section 3.4.2) or make use of the Database class to access multiple samples from the current project (as explained in Section 3.4.3).

All Viper modules must comply with a specific structure to be recognised by the framework (Guarnieri, 2015a). New modules must inherit from the abstract Module class and declare command and description properties. Additionally, all modules must implement a `run()` function, as shown in Listing 3.12.

```
from viper.core.database import Database

class Decode_All(Module):
    cmd = 'decode_all'
    description = 'Reveals code hidden by eval() or preg_replace() constructs
      in all samples'

    def run(self):

        # Get Viper's root path
        viper_path = get_viper_path(__project__.get_path())

        # Retrieve all the samples from the database
        db = Database()
        samples = db.find(key='all')

        # Decode all samples
        for sample in samples:
            ...

```

Listing 3.11: Decode_All.py extract demonstrating the use of the `find()` function

```
from viper.common.abstracts import Module

class NewModule(Module):
    cmd = 'new_module'
    description = 'new_module description'

    def run(self):
      ...

```

Listing 3.12: Structure of a generic Viper module

3.5 Individual Modules

Four preprocessing modules were created to process samples in different ways to prepare them for similarity analysis. Each of these modules was designed to be run against a single shell sample, and require that a Viper session already exists (see Section 3.4.2 for more information on sessions in Viper). Both HashChunks.py and HtmlDump.py process samples in their entirety and produce a new file, whereas Functions.py and FunctionBodies.py extract and store relevant features for analysis. A list of all the individual modules, their associated commands, and a short description of their functionality is displayed in Table 3.4.

All but one of the individual modules make use of external PHP scripts to carry out their tasks. This is due to the presence of functions and extensions in the core PHP library that directly support the deconstruction and analysis of PHP code. A good example of this is the Tokenizer

Table 3.4: Individual modules and their descriptions

Module	Command	Description
FunctionBodies.py	`bodies`	Extracts user-defined function bodies from an sample
Functions.py	`functions`	Creates a list of user-defined functions for an open sample
HashChunks.py	`chunks`	Splits an open sample into fixed-size chunks and hashes each chunk
HtmlDump.py	`html_dump`	Generates and stores an HTML dump for an open sample

extension, which is used by both the Functions.php and FunctionBodies.php helper scripts to extract the names and bodies of user-defined functions on behalf of their Viper parent modules (Functions.py and FunctionBodies.py, respectively).

Interactions between the Viper modules and their helper scripts are facilitated by Python's subprocess module, which allows for the creation and execution of separate child processes (Python Software Foundation). The module's `check_output()` function allows a Python script to run a command with a list of arguments and capture the output as a byte string. This string can then be written to file by the parent module for use during the similarity analysis process.

The `check_output()` function accepts a list of command line arguments that are used to start a new process. The list of arguments used by the individual modules to start an external PHP script is given in Listing 3.13.

```
subprocess.check_output(['php', '-f',
  viper_path + '/modules/FunctionBodies.php', path])
```

Listing 3.13: List of arguments passed to the `check_output()` function

The path to the helper script in Listing 3.13 is determined by appending an appropriate extension to the root Viper installation path (in this case '/modules/FunctionBodies.php'), where the `'path'` argument represents the absolute path to the sample that the script should be run against. The root installation path is retrieved by calling the `get_viper_path()` function that was added to Viper's utils script, while the path to the sample is retrieved by accessing the current session object, as shown in Listing 3.14.

Along with all of the other Viper extension modules developed as part of the similarity analysis system, the individual modules all include comprehensive usage instructions. This is in keeping with the goal outlined in Section 3.1, which is to integrate the current similarity analysis system for PHP files into the existing Viper framework. Figure 3.6 shows an example of the usage instructions for the Dendrogram.py visualisation module.

```
1 ...
2
3 # Get Viper's root path
4 viper_path = get_viper_path(__project__.get_path())
5
6 # Get the file path for use by the FunctionBodies.php script
7 path = __sessions__.current.file.path
8
9 ...
```

Listing 3.14: Code extract demonstrating the retrieval of the root Viper path and sample path

Figure 3.6: Example of the help dialogue for the Dendrogram.py visualisation module

3.5.1 FunctionBodies.py

The purpose of the FunctionBodies.py module is to extract the contents of all user-defined function bodies present in a malware sample for subsequent comparative analysis. The identification and extraction of these bodies required that the samples be separated into tokens, which was more easily achieved using PHP itself. For this reason, the FunctionBodies.py module makes use of an external PHP script, as shown on lines 5 and 6 of Listing 3.15. Lines 7 and 8 demonstrate how the output returned from the external script is appended with the `'(bodies)'` identifier and written to file alongside the original.

The external FunctionBodies.php script makes use of PHP's Tokenizer extension (which is discussed in more detail in Section 2.8.1) to identify and extract the bodies of user-defined functions (PHP Group, 2015h). At the start, the source code is split into a stream of PHP language tokens and basic characters using the `token_get_all()` function, which interfaces with the Zend engine's lexical scanner. These tokens and characters are consumed one by one until a

```
1  ...
2
3  # Call the FunctionBodies.php script and store its output
4  try:
5    output = subprocess.check_output(['php', '-f',
6      viper_path + '/modules/FunctionBodies.php', path])
7    f = open(path + '(bodies)', 'w')
8    f.write(output)
9    f.close()
10   print_success('Complete')
11 except subprocess.CalledProcessError:
12   print_error('Failed to reach the FunctionBodies.php script')
13
14 ...
```

Listing 3.15: FunctionBodies.py extract demonstrating the interaction with the FunctionBodies.php helper script

T_FUNCTION token is found. Once a this token has been found, tokens and characters are consumed until an opening curly bracket character is found, signifying the start of a function body. Thereafter, curly brackets are counted (and tokens and characters recorded) until the final closing curly bracket is reached, at which point the body is returned to the FunctionBodies.py module. The implementation of this logic is shown as pseudo-code in Listing 3.16.

3.5.2 Functions.py

The Functions.py module is similar to the FunctionBodies.py module, but it extracts only the names of any user-defined functions and ignores their associated bodies. As was the case with the FunctionBodies.py module, this feature extraction process is performed by an external PHP script, as shown in lines 5 and 6 of Listing 3.17. Lines 7 and 8 demonstrate how the output returned from the external script is appended with the `'(functions)'` identifier before being written to file alongside the original.

The Functions.php helper script also makes use of PHP's built in Tokenizer extension to identify and extract the names of user-defined functions from a given sample (PHP Group, 2015h). Firstly, a stream of PHP language tokens and basic characters is created from the source code using the `token_get_all()` function. These tokens and characters are then consumed until a `T_FUNCTION` token is found, indicating the start of a function definition. Once a `T_FUNCTION` token has been found, the next `T_STRING` token or basic character is returned to the Functions.py parent module as the name of the user-defined function. The full pseudo-code of the Functions.php helper script is shown in Listing 3.18.

```
BEGIN
  Split sample into PHP tokens and characters
  FOR every token or character
    IF it is a token
      IF FoundStarting
        Add token to body
      ELSE IF the token is a function
        FoundFunction is true
      END IF
    ELSE
      IF FoundFunction but not FoundStarting and character is '{'
        FoundFirstBracket is true
        BracketCount is 1
        Add character to body
      ELSE IF FoundFunction and FoundStarting and character is '{'
        BracketCount = BracketCount + 1
        Add character to body
      ELSE IF FoundFunction and FoundStarting and character is '}'
        BracketCount = BracketCount - 1
        Add character to body
      ELSE IF FoundFunction and FoundStarting and character isn't '{' or '}'
        Add character to body
      END IF

      IF BracketCount is 0
        Output body to Decode.py
        Reset FoundFunction, FoundStarting, and BracketCount
      END IF
    END IF
  END FOR
END
```

Listing 3.16: Psuedo-code describing the logic implemented in the FunctionBodies.php helper script

3.5.3 HashChunks.py

The purpose of the HashChunks.py module is to separate a file into chunks of equal length and to hash each chunk using Ssdeep, an algorithm capable of producing fuzzy hashes (see Section 2.9.4 for more information on fuzzy hashing). Unlike the Functions.py and FunctionBodies.py modules, it does not rely on a PHP helper script to perform this task. This is due to the existence of Pydeep, a collection of Python bindings for the Ssdeep library, which was originally written in C. Section 2.10 provides a comprehensive description of both Ssdeep and the Pydeep Python wrapper.

Listing 3.19 shows an extract from the HashChunks.py module that demonstrates the separation and hashing of a given sample. At first, unnecessary formatting is removed from the code to reduce the number of chunks needed for each file, thereby speeding up the hashing process. Because this modification is applied uniformly to all samples before any analysis takes place, it has no effect on the similarity scores calculated by the Matrix.py module (which is described in

```
...

# Call the Functions.php script and store its output
try:
  output = subprocess.check_output(['php', '-f',
    viper_path + '/modules/Functions.php', path])
  f = open(path + '(functions)', 'w')
  f.write(output)
  f.close()
  print_success('Complete')
except subprocess.CalledProcessError:
  print_error('Failed to reach the Functions.php script')

...
```

Listing 3.17: Functions.py extract demonstrating the interaction with the Functions.php helper script

```
BEGIN
  Split sample into PHP tokens and characters
  FOR every token or character
    IF it is a token
      IF the token is a function
        FoundFunction is true
      ELSE IF FoundFunction and the token is a string
        Output the string to Functions.py
        FoundFunction is false
      END IF
    ELSE IF FoundFunction
      Output the character to Functions.py
      FoundFunction is false
    END IF
  END FOR
END
```

Listing 3.18: Psuedo-code describing the logic implemented in the Functions.php helper script

more detail in Section 3.7).

Once the sample has been stripped of all whitespace characters, it is divided into chunks of 200 characters each. This value was chosen as a result of a documented limitation of the Pydeep library's `compare()` function that causes it to assign incorrect similarity scores to identical chunks when the lengths of the chunks fall below 193 characters (refer to Section 2.10.4 for a description of this effect). Each chunk is then hashed using Pydeep's `hash_buf()` function before being written to file for future use.

```
1  ...
2
3  # Remove whitespace
4  contents = contents.replace(' ', '')
5  contents = contents.replace('\t', '')
6  contents = contents.replace('\n', '')
7
8  # Split into chunks
9  chunks = [contents[i:i+200] for i in range(0, len(contents), 200)]
10
11 # Hash every chunk and write it to file
12 for chunk in chunks:
13   hashed_chunk = pydeep.hash_buf(chunk)
14     f.write(hashed_chunk + '\n')
15 f.close()
16 print_success('Complete')
17
18 ...
```

Listing 3.19: HashChunks.py extract demonstrating how each sample is separated and hashed

3.5.4 HtmlDump.py

The HtmlDump.py module is used to execute a given sample and record the HTML that it produces for future analysis. This is trivially achieved by including the file in a PHP script (using the `include()` statement) and monitoring the output. The HtmlDump.py module therefore makes use of the external HtmlDump.php script to run a shell, store its output in a buffer, and then return this buffer to the parent Python module, as demonstrated in Listing 3.20 in lines 5 and 6. Lines 7 and 8 show how the output obtained from the external script is appended with the `'(html)'` identifier before being written to file alongside the original sample.

```
1  ...
2
3  # Call the HtmlDump.php script and store its output
4  try:
5    output = subprocess.check_output(['php', '-f',
6      viper_path + '/modules/html_dump.php', path])
7    f = open(path + '(html)', 'w')
8    f.write(output)
9    f.close()
10   print_success('Complete')
11 except subprocess.CalledProcessError:
12   f = open(path + '(html)', 'w')
13   f.write('failed')
14   f.close()
15
16 ...
```

Listing 3.20: HtmlDump.py extract demonstrating the interaction with the HtmlDump.php helper script

As can be seen in lines 11 to 14, the exception clause associated with the call to the external HtmlDump.php script differs from those present in the other individual modules. This is because several of the samples used during the testing process contained malformed HTML code that would cause the `include()` statement to throw an exception, which would be stored in the buffer and returned to the HtmlDump.py parent module. To avoid these exceptions being compared against syntactically correct HTML during the similarity analysis process (thereby influencing the results), a failure message is stored in its place. Section 3.7.3 demonstrates how HTML failure messages are handled during similarity analysis.

```
1  ...
2
3  ob_start();
4
5  include($path);
6
7  echo ob_get_contents();
8
9  ob_end_clean();
10
11 ...
12 }
13
```

Listing 3.21: HtmlDump.php extract demonstrating the output buffering process

Listing 3.21 demonstrates how the HtmlDump.php helper script uses PHP's output control functions (PHP Group) to process a sample and buffer its output before returning it to the parent module. Output buffering is enabled using the `ob_start()` function in line 3. After inclusion and processing of the sample file, the content of the buffer is retrieved and echoed to the Python parent module via the `ob_get_contents()` function in line 7. Finally, the buffer is cleared and output buffering is disabled using the `ob_end_clean()` function in line 9.

3.6 Batch Modules

The batch modules contain no feature extraction or sample processing capabilities of their own, but rather apply each of the individual modules to all the samples in the current project (see Section 3.4.1 for more information on projects in Viper). The purpose of the batch modules is to prepare an entire collection of samples for comparison by the Matrix.py module. Each of the command line options contained in this comparison module (apart from a special case involving unprocessed samples) require that a specific batch module already be complete. A list of the batch modules and a short description of their functionality is given in Table 3.5.

Each of the batch modules can be run against either raw or decoded shell samples, with the exception of HtmlDumpAll.py. The target set can be specified by appending the desired command

Table 3.5: Batch modules and their descriptions

Module	Command	Description
FunctionBodiesAll.py	`bodies_all`	Extracts user-defined function bodies from all samples
FunctionsAll.py	`functions_all`	Creates a list of functions for all samples
HashChunksAll.py	`chucks_all`	Splits all samples into fixed-size chunks and hashes each chunk
HtmlDumpAll.py	`html_dump_all`	Generates and stores an HTML dump for all samples

with either `'-r'` (for the raw set) or `'-d'` (for the decoded set). If the decoded set is selected, the module first determines whether all samples have been decoded before proceeding with the batch processing. This test is conducted by the `all_decoded()` function, which searches for the presence of decoded files in the Viper repository (see Listing 3.22).

The batch modules originally included options to generate HTML dumps for both raw and decoded sample sets, but this was found to be unnecessary. During testing it was discovered that both raw and decoded versions of the same sample always generated the same HTML output, and the ability to target both sample sets was thus removed from the system.

```
1  def all_decoded(self, sample_names):
2    for sample in sample_names:
3      path = get_sample_path(sample) + '(decoded)'
4      try:
5        f = open(path)
6        f.close()
7      except IOError:
8        print_error('Not all samples have been decoded.
9          Run the decode_all command and then try again')
10       return False
11   return True
```

Listing 3.22: Implementation of the `all_decoded()` function

3.7 Matrix Module

The purpose of the Matrix.py module is to produce matrices that represent the observed similarity between all samples in a given collection based on a specified metric. This module relies on the feature extraction and sample processing performed by the aforementioned batch functions (which in turn rely on the individual functions to perform their tasks). In addition to creating and populating similarity matrices, the Matrix.py module also produces a list of sample labels. These labels are written to file for later use as axis markers by the Heatmap.py and Dendrogram.py visualisation modules.

Several options can be passed to the Matrix.py module. Each option represents the measure of similarity that should be used to generate a similarity matrix. If one would like to view

the similarity of user-defined function names between raw shells in a project, for example, the command would be `'matrix -f raw'`. To make use of the same measure of similarity (i.e., function name matches) on decoded shells in a project, the command would be `'matrix -f decoded'`. A full list of the available option combinations is given in Table 3.6.

Table 3.6: Possible option combinations for Matrix.py

Options	Description
-b raw	Compares the function bodies of raw samples
-b decoded	Compares the function bodies of decoded samples
-f raw	Compares the function names of raw samples
-f decoded	Compares the function names of decoded samples
-c raw	Compares the hashed chunks of raw samples
-c decoded	Compares the hashed chunks of decoded samples
-H	Compares the HTML dumps generated by samples

3.7.1 Preliminary Setup

The Matrix.py module performs some preliminary tasks regardless of the chosen option combination. These steps were separated from the different matrix creation processes to avoid any unnecessary repetition of code and are given in Listing 3.23.

First, the path to the root Viper installation is determined to act as a reference point for all file I/O operations performed during matrix creation. As discussed in Section 3.1, the system was designed with other users in mind; for this reason, lines 6 to 13 check for the existence of the 'data/matrix' and 'data/labels' directories, which are required for matrix creation but do not form part of the default Viper installation. If these directories are not found, they are created by appending the appropriate extension to the root Viper path.

The name of the current Viper project is used to link the project with its associated matrix and label files. Lines 15 to 18 attempt to retrieve the project name from the global `__project__` object. If no name is found, Viper's default project is active, and an empty string is assigned to the `'project'` variable.

The remainder of Listing 3.23 (lines 20 through 33) prepares a list of samples for comparison and allocates an appropriately sized two-dimensional array to store the resulting matrix. This is done by using Viper's Database class to access all project samples via the `find()` function before creating a list of sample names from the resulting object list. The length of this list is used to create the two-dimensional array.

```
1  ...
2
3  # Get Viper's root path
4  viper_path = get_viper_path(__project__.get_path())
5
6  # Check that the label and matrix folders exist
7  directory = viper_path + '/data/labels'
8  if not os.path.exists(directory):
9    os.makedirs(directory)
10 directory = viper_path + '/data/matrix'
11 if not os.path.exists(directory):
12   os.makedirs(directory)
13
14 # Get the project name
15 project = __project__.name
16 if project is None:
17   project = ''
18
19 # Retrieve all the samples from the database
20 db = Database()
21 samples = db.find(key='all')
22
23 # Count the number of samples
24 total = len(samples)
25
26 # Create a list of samples
27 sample_names = []
28 for sample in samples:
29   sample_names.append(str(sample.sha256))
30
31 # Create an appropriately-sized matrix
32 matrix = [[0 for x in range(total)] for x in range(total)]
33
34 ...
```

Listing 3.23: Matrix.py extract demonstrating the preliminary setup tasks

3.7.2 Matrix Creation

Once a list of sample names and an appropriately-sized matrix have been created during preliminary setup, the matrix is populated by comparing every sample in a given project to every other sample using a specified measure of similarity. This process varies slightly based on the combination of options passed to the Matrix.py module during invocation. Listing 3.24 demonstrates how a matrix is computed when the `'-b raw'` option combination is selected.

First, several tracking variables are created to monitor statistics relating to the similarity analysis process. As can be seen in lines 17 through 23, the `count`, `total_similarity`, `mini`, and `maxi` variables are updated each time a match of any magnitude is found. The maximum value in this case is taken to be the greatest value detected between two unique samples, and the update check for a new maximum match thus excludes cases where a sample is being compared to itself. This was implemented by appending the `'x != y'` check to the usual maximum test

```
1  ...
2
3  if arg_bodies == 'raw':
4
5    # Populate the similarity matrix
6    count = 0
7    total_similarity = 0
8    mini = matrix[0][0]
9    maxi = matrix[0][0]
10   for x in range(0, total):
11     path1 = get_sample_path(sample_names[x])
12
13     for y in range(0, total):
14       path2 = get_sample_path(sample_names[y])
15       matrix[x][y] = self.compare_bodies(path1, path2)
16
17       if(matrix[x][y] > 0):
18         total_similarity += matrix[x][y]
19           if(matrix[x][y] < mini):
20             mini = matrix[x][y]
21           if(matrix[x][y] > maxi and x != y):
22             maxi = matrix[x][y]
23           count += 1
24
25   # Save the matrix for later use
26   numpy.save(viper_path + '/data/matrix/bodies(raw)(' + project + ')', matrix)
27
28   # Display the matrix
29   arr = numpy.array(matrix)
30   print(numpy.flipud(arr))
31
32 ...
```

Listing 3.24: Matrix.py extract demonstrating matrix creation for the `'-b raw'` option combination

in line 21.

The algorithm then compares each sample against every other sample by determining the paths to the samples and passing them to the `compare_ssdeep()` comparison function. The value that is returned by this function is stored in the matrix, which is written to file and displayed to the user.

The process outlined in Listing 3.24 is identical for all option combinations targeting raw sample sets, with the exception of lines 15 and 26. Each measure of similarity that forms part of the Matrix.py module is associated with its own comparison function, all of which are described in more detail in Section 3.7.3. The `compare_ssdeep()` function in line 15 is thus substituted for an appropriate comparison function depending on the chosen measure of similarity (i.e., `compare_bodies()` for function body matching, `compare_chunks()` for matching hashed chunks and so on). The file name of the matrix storage file in line 26 also differs for each option combination to facilitate the storage of multiple matrices at one time.

3.7.3 Comparison Functions

The comparison functions are used to calculate the observed similarity between two given files. As described in the previous section, a completed matrix represents the collation of the results returned by a comparison function for every pair of samples in the project. All values returned by the comparison functions represent a similarity percentage and thus lie in the range from zero to 100 inclusive.

compare_chunks()

The `compare_chunks()` function takes two lists of hashed file chunks as its input and calculates an overall similarity value for them. As was the case with `compare_ssdeep()`, it makes use of the Pydeep library's `compare()` function to compare fuzzy hashes, although in this case the process needs to be repeated for each hash in the list before an average value can be calculated. The signature and implementation of the `compare_chunks()` function is shown in Listing 3.25.

```
1  def compare_chunks(self, file1, file2):
2
3    # Open the hashed chunk lists for each file
4    with open(file1 + '(chunks)') as f1:
5      contents1 = f1.readlines();
6    with open(file2 + '(chunks)') as f2:
7      contents2 = f2.readlines();
8
9    # Select the longest list as the total
10   count = max(len(contents1), len(contents2))
11   if(count == 0):
12     return 0
13
14   # Count the hashed chunk matches
15   matches = 0
16   for linehash1 in contents1:
17     highest = 0
18     for linehash2 in contents2:
19       current_match = pydeep.compare(linehash1, linehash2)
20       if(current_match > highest):
21         highest = current_match
22     matches += highest
23
24   return int(matches/count)
```

Listing 3.25: Signature and implementation of the `compare_chunks()` function

To start with, the hashes originally created by the HashChunksAll.py batch module are read into two lists. The length of the longest list is then chosen as the count that is later used to calculate the average similarity value in line 24. The longer length is used as the divisor in this case to

prevent samples of unequal length being declared exactly similar. As example of this, consider files A and B in Figure 3.7. The first five characters of each of these files match exactly, and if each exact match were to be assigned a value of 100, the total similarity for the two files would be calculated as 500 by the algorithm outlined in Listing 3.25. If one were to divide this by the number of characters in file B, the overall similarity value for the two files would be 100, indicating a perfect match, which is clearly not the case given that the two files are of different length. Dividing the total by the number of characters in file A returns the correct overall similarity value of 55.55.

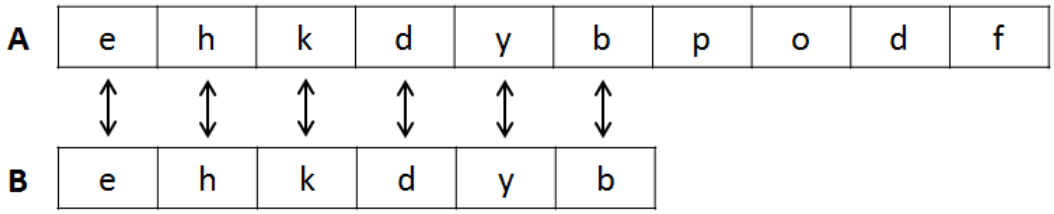


Figure 3.7: Comparing files of unequal length

Lines 14 to 22 compare each hash in the first list to every hash in the second. To prevent cross-chunk similarity from affecting the final value (i.e., to prevent the function from returning a value greater than 100%), only the highest similarity is recorded in each case. As an example of the effects of cross-chunk similarity, consider the two lists of hashes A and B shown in Figure 3.8. If hash B1 were to be compared to each of the hashes in list A, and each of the calculated similarity values were added together, B1 would register a similarity value of 250%, which is clearly not an accurate figure. The algorithm detailed in Listing 3.25 therefore ensures that only the highest match (in this case 80%) is recorded as the similarity value. Once the total similarity has been calculated in this way, it is divided by the length of the longer of the two lists in order to determine an overall similarity value for the two files.

compare_funcs()

The `compare_funcs()` function calculates the average similarity between two lists of function names. This is achieved by determining which names occur in both lists and returning this figure as a percentage of the function list with the most names. The simple implementation of the `compare_funcs()` function is shown in Listing 3.26

Unlike the `compare_chunks()` function described in the previous section, `compare_funcs` compares function names exactly (i.e., without resorting to fuzzy hashing), as is apparent from

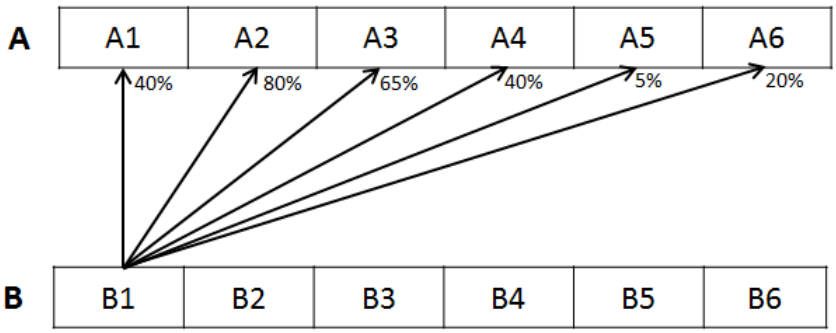


Figure 3.8: Illustration of cross-chunk similarity

the presence of the `'in'` operator in line 17. The use of fuzzy hashing in the `compare_chunks()` function was precipitated by a need to determine the similarity between large sample chunks, whereas in the case of `compare_funcs()` the presence of an exact function name is considered a mark of similarity.

Since PHP does not support function overloading, it is not necessary to cater for multiple function definitions with the same name. As discussed in Section 3.5.2, the lists generated by the Functions.py module are guaranteed to be unique – each function name will therefore match once or not at all.

compare_bodies()

The `compare_bodies()` function uses fuzzy hashing to detect subtle changes in function implementations. Like the `compare_chunks()` function, it uses the Pydeep Python library to hash a feature extracted by the appropriate individual or batch module (in this case FunctionBodies.py or FunctionBodiesAll.py, respectively) and then compares these hashes to calculate an overall level of similarity for two given samples. Listing 3.27 details the implementation of the `compare_bodies()` function.

The FunctionBodies.py and FunctionBodiesAll.py modules both separate multi-line function bodies with the '####BREAK####' string to allow them to easily be split into lists of individual bodies (as is performed in Listing 3.27 on lines 10 and 11). These bodies are then hashed and compared to one another using Pydeep's `hash_buf()` and `compare()` functions. As was the case with the `compare_chunks` function, only the highest match for each function implementation is added to the total, which is then used to calculate and return an average level of body similarity.

```
1  def compare_funcs(self, file1, file2):
2
3    # Open the function name lists for each file
4    with open(file1 + '(functions)') as f1:
5      contents1 = f1.readlines();
6    with open(file2 + '(functions)') as f2:
7      contents2 = f2.readlines();
8
9    # Select the longest list as the total
10   count = max(len(contents1), len(contents2))
11   if(count == 0):
12     return 0
13
14   # Count the function name matches
15   matches = 0
16   for func_name in contents1:
17     if func_name in contents2:
18       matches += 1
19
20   return int(round((matches/count) * 100))
```

Listing 3.26: Signature and implementation of the `compare_funcs()` function

compare_html()

The purpose of the `compare_html()` function is to use fuzzy hashing to determine the level of similarity between two HTML files. As such it requires access to the files generated by either the HtmlDump.py or HtmlDumpAll.py modules. The simple function signature and implementation of the `compare_html` function is shown in Listing 3.28.

As described in Section 3.5.4, some of the malware samples used during testing contained malformed HTML that caused the HtmlDump.py and HtmlDumpAll.py modules to register an exception and prevented them from writing the HTML to file. Because of this, the two modules were configured to record the failure in the files originally intended for the HTML. The `compare_html()` function therefore tests for occurrences of failed HTML recording attempts (as seen in line 10) and returns a similarity value of zero if one is detected.

3.7.4 Validation Functions

Each option combination in the Matrix.py module is associated that a validation function which ensures that the batch functions needed to create the required files have been run successfully. As discussed in Section 3.1, the system was developed with other users in mind, and not just for this research. These validation functions provide a simple means of determining the existence of the files needed for matrix creation, and allow Viper to print a helpful error message instead of crashing. As an example, Figure 3.9 demonstrates the error message that is displayed when a

```
1  def compare_bodies(self, file1, file2):
2
3    # Open the function body files
4    with open(file1 + '(bodies)') as f1:
5      contents1 = f1.read();
6    with open(file2 + '(bodies)') as f2:
7      contents2 = f2.read();
8
9    # Split the function body files into lists of function bodies
10   bodies1 = contents1.split('####BREAK####')
11   bodies2 = contents2.split('####BREAK####')
12
13   # Select the longest list as the total
14   count = max(len(bodies1), len(bodies2))
15   if(count == 0):
16     return 0
17
18   # Count the function body matches
19   matches = 0
20   for body1 in bodies1:
21     highest = 0
22     for body2 in bodies2:
23       current_match = pydeep.compare(pydeep.hash_buf(body1), pydeep.hash_buf(body2))
24       if(current_match > highest):
25         highest = current_match
26     matches += highest
27
28   return int(matches/count)
```

Listing 3.27: Signature and implementation of the `compare_bodies()` function

matrix representing the similarity between function names in a set of raw samples is requested but the `'functions_all -r'` command has yet to be run. Table 3.7 provides a complete list of all the error messages associated with each validation function.

The implementations of the various validation functions are based on the simple `all_decoded()` function described in Section 3.6 – only the file path and error messages change in each case. Because of this, Listing 3.29 only provides details of the `all_bodies_raw()` function as an example, as all the other validation functions can be extrapolated from this implementation.

3.8 Visualisation Modules

The purpose of the visualisation modules is to create a graphical representation of a given similarity matrix. Although these representations don't contain exact matching values, they are easier to interpret, and can be studied to discover relationships between samples.

Both the of the visualisation modules were implemented using a combination of the data visualisation tools described in Section 2.11.4. Numpy arrays are used to store the matrices prior

```
def compare_html(self, file1, file2):

  # Open the HTML dumps for each file
  with open(file1 + '(html)') as f1:
    contents1 = f1.read();
  with open(file2 + '(html)') as f2:
    contents2 = f2.read();

  # If either of the HTML dumps is malformed, return zero
  if(contents1 == 'failed' or contents2 == 'failed'):
    return 0

  return pydeep.compare(pydeep.hash_file(file1 + '(html)'), pydeep.hash_file(file2 + '(html)'))
```

Listing 3.28: Signature and implementation of the `compare_html()` function

Figure 3.9: Request for a function name matrix when the `'functions_all -r'` command has yet to be run

to rendering, and the matplotlib library is used to create both the heatmap and dendrogram objects. In addition to this, Dendrogram.py module makes use of the hierarchical clustering capabilities of the SciPy library to group samples before they are displayed.

3.8.1 Heatmap.py

The Heatmap.py module is used to display each value in a given matrix as a colour that represents the magnitude of that value. Heatmaps can be generated from matrices created using any of the measures of similarity listed in Table 3.6. As is discussed in Section 2.11.2, clusters of dark colours represent areas of greater similarity, while lighter areas indicate a lack of similarity. The brief implementation of the Heatmap.py module's `draw_heatmap()` function is outlined in Listing 3.30.

Table 3.7: Validation functions and their associated error messages

Validation Function	Error Message
all_chunked_raw	Not all raw samples have been hashed. Run the chunks_all command and then try again.
all_chunked_decoded	Not all decoded samples have been hashed. Run the chunks_all command and then try again.
all_functions_raw	Not all raw samples have had their function names extracted. Run the functions_all command and then try again.
all_functions_decoded	Not all decoded samples have had their function names extracted. Run the functions_all command and then try again.
all_bodies_raw	Not all raw samples have had their function bodies extracted. Run the bodies_all command and then try again.
all_bodies_decoded	Not all decoded samples have had their function bodies extracted. Run the bodies_all command and then try again.
all_html_raw	Not all raw samples have had their HTML extracted. Run the html_dump_all command and then try again.

```
1  def all_bodies_raw(self, sample_names):
2    for sample in sample_names:
3      path = get_sample_path(sample) + '(raw)(bodies)'
4      try:
5        f = open(path)
6        f.close()
7      except IOError:
8        print_error('Not all raw samples have had their function bodies extracted.
9          Run the bodies_all command and then try again')
10       return False
11   return True
```

Listing 3.29: Implementation of the `all_bodies_raw()` validation function

3.8.2 Dendrogram.py

As is described in Section 2.11.3, dendrograms are tree-like structures that can be used to display relationships that result from hierarchical clustering algorithms. Dendrogram.py performs this clustering and displays the resulting figure, and can be run on any matrix created using the measures of similarity listed in Table 3.6. The hierarchical nature of the dendrograms produced in this way allows for the identification of derivative sample relationships, as well as the magnitude of such relationships. Listing 3.31 details the implementation of the Dendrogram.py module's `draw_dendrogram()` function.

```
1  import numpy
2  import json
3  import matplotlib.pyplot as plot
4  ...
5  def draw_heatmap(self, matrix, labels_path):
6    # Get labels and set the label size
7    with open(labels_path, 'r') as infile:
8      sample_names = json.load(infile)
9    lsize = 100/len(sample_names)
10
11   # Truncate long sample names
12   for ind in range(len(sample_names)):
13     if len(sample_names[ind]) > 8:
14       sample_names[ind] = sample_names[ind][0:8]
15
16   # Set up and display the heatmap
17   fig, ax = plot.subplots()
18   heatmap = ax.pcolor(matrix, cmap=plot.cm.Blues)
19   plot.show()
20 ...
```

Listing 3.30: Implementation of the `draw_heatmap()` function

3.9 Chapter Summary

This chapter discussed the various components of the system responsible for the detection of derivative PHP-based malware samples. A high-level overview of the system's structure was provided, including an description of how test samples are passed from one component to another. The two download scripts responsible for retrieving and storing these files were then introduced, as was the decoder component, which is used to deobfuscate and normalise sample inputs prior to analysis. Viper, the malware analysis framework that was used as the basis for the similarity analysis system, was described thereafter. This was followed by an outline of the individual and batch modules that were used to extract pertinent features for subsequent comparison by the matrix module. The chapter concluded with a description of the two modules that were used to create visualisations of the results produced by the matrix module for the purposes of identifying meaningful inter-sample relationships.

```
1  import numpy
2  import json
3  import pylab
4  import scipy.spatial.distance as ssd
5  import scipy.cluster.hierarchy as sch
6  ...
7  def draw_dendrogram(self, matrix, labels_path):
8    # Get labels and set the label size
9    with open(labels_path, 'r') as infile:
10     sample_names = json.load(infile)
11   lsize = 100/len(sample_names)
12
13   # Truncate long sample names
14   for ind in range(len(sample_names)):
15   if len(sample_names[ind]) > 8:
16   sample_names[ind] = sample_names[ind][0:8]
17
18   # Set up and display the dendrogram
19   fig = pylab.figure()
20   matrix = ssd.pdist(matrix)
21   linkage_matrix = sch.linkage(matrix)
22   sch.dendrogram(linkage_matrix, labels=sample_names, leaf_font_size=15)
23   pylab.xlabel('Samples', size=15)
24   pylab.ylabel('Distance', size=15)
25
26   fig.show()
27 ...
```

Listing 3.31: Implementation of the `draw_heatmap()` function

4

Results

This chapter begins with a description of the collection of samples that was used for testing purposes in Section 4.1. It then goes on to evaluate the effectiveness of the Decoder.py module and its attempts to normalise and deobfuscate malicious samples in Section 4.2, before moving on to test the functionality of each of the individual modules in Section 4.4. Section 4.6 demonstrates the capabilities of the visualisation modules by conducting an in-depth study of the `c99` family of shells. These capabilities are then applied to more comprehensive shell collections in Section 4.7. The chapter concludes with an evaluation of the four primary similarity measures of similarity in Section 4.8.

Throughout the testing process the performance of several key components was evaluated to ensure that they were able to process large datasets within a reasonable time period. This is in keeping with the goal (outlined in Section 3.1) of eventually integrating parts of this research into the Viper malware analysis framework described in Section 3.4.

4.1 Test Data

A total of 2 129 functional RATs were collected during this research for use as inputs to the system. These test samples were obtained from a variety of online sources, all of which are listed in Table 4.1.

Table 4.1: Sample source breakdown

Source	**Number of Shells**
VirusTotal.com	1969
Insecurety.net	87
c99shell.gen.tr	21
r57shell.net	7
r57.gen.tr	10
hoco.cc	35
Total	2129

The vast majority of the samples in the test collection were obtained from a repository maintained by the VirusTotal online malware analysis service discussed in Section 2.7. These samples were identified and retrieved through the service's private API by the download scripts (Section 3.2) using a parametrised query that was targeted only those malware samples that matched the criteria for this research (i.e., RATs written in PHP). As an additional requirement, each sample had to have been flagged as malicious by at least one of the 54 antivirus engines available on the VirusTotal platform.

In addition to the automated retrieval of malware samples from the VirusTotal repository, several of the more more popular shells were manually sourced from a number of online collections. The largest of these sources was a web malware archive curated by Insecurety Research[1], which contained a variety of malicious scripts written in PHP. This archive was supplemented by smaller collections obtained from the four remaining sources listed in Table 4.1. The inclusion of multiple sample sources created the possibility of redundancy, as several of the more common RATs were found in more than one collection. For this reason, an MD5 hash was generated for each file and compared to the hashes of every other file to ensure that no two samples were identical.

The 2 129 collected samples ranged in size from 34B to 726kB, with the average shell containing 44.3kB of data. A histogram of these file sizes is shown in Figure 4.1. Some of the larger shells included sections of embedded HTML to create user-friendly GUIs similar to the 'N3tShell' example given in Figure 4.2. These simple GUIs allow users with limited technical expertise to access and make use of advanced remote capabilities such as kernel fingerprinting, code execution, and file manipulation with relative ease.

A Note on Sample Naming Conventions

As a result of the informal nature of many of the online shell sources, as well as active attempts on the part of malware authors to protect their code, several of the file names associated with the malware samples were found to be inconsistent and unrelated to the contents of the files. The presence of several variants of the same shell also meant that some file names were often

[1] http://insecurety.net/?p=96

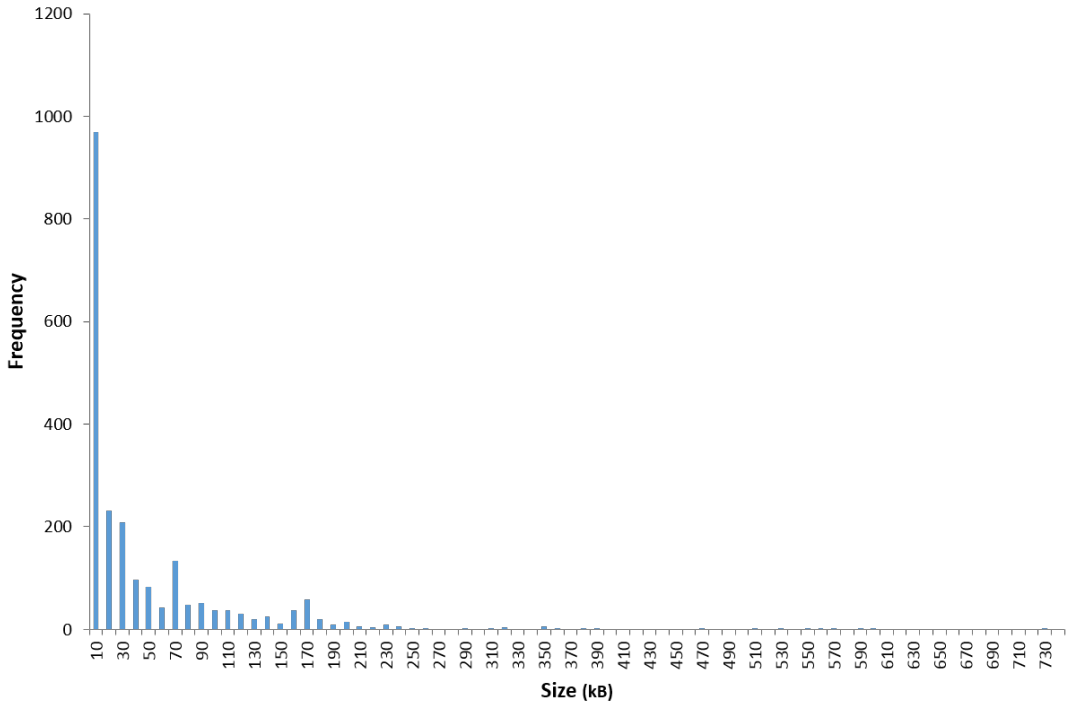


Figure 4.1: Histogram of file sizes for the test collection

duplicated. Despite this, a decision was made to use these assigned file names where possible throughout this chapter, as they were found to contribute more in the way of semantic meaning than other contrived naming conventions such as the generation of a hash signature for each shell. When examining results produced during similarity analysis, the presence of similar file names was not used as a guarantee of similarity, but rather as an incentive for closer inspection. The malware samples retrieved from the collection maintained by VirusTotal were named using their MD5 hashes, as the service does not store descriptive file names. In keeping with the conventions of the Viper framework discussed in Section 3.4, all samples were stored and retrieved using their SHA256 hashes.

4.2 Decoder Tests

The extensible decoder component described in Section 3.3 is responsible for performing code normalisation and deobfuscation prior to similarity, with the goal of exposing more code for analysis. As such, it can be declared a success if it is able to remove all layers of obfuscation from a script (i.e., if it removes all `eval()` and `preg_replace()` constructs). The tests for this component progressed from scripts containing simple, single-level `eval()` and `preg_replace()` statements to more comprehensive tests involving auxiliary functions and nested obfuscation constructs. Each test was designed to clearly demonstrate a specific capability of

Figure 4.2: GUI of a derivative of the popular `c99` shell

the decoder. Finally, a test was performed with a fully-functional web shell as the input to demonstrate the decoder's performance with live samples.

4.2.1 Single-level Eval() and Base64_decode()

The most basic test of the decoder involved providing a single `eval()` statement and base64-encoded argument as input and recording whether it was correctly identified, extracted, and replaced with the code that it was obscuring. The constructed test script is shown in Listing 4.1.

```
1 <?php
2   echo "Hello";
3   eval(base64_decode("ZWNobyAiR29vZGJ5ZSI7"));
4 ?>
```

Listing 4.1: Single-level `eval()` with a base64-encoded argument

To create the input script, a simple `echo()` statement (with 'Goodbye' included as an argument) was encoded using PHP's `base64_encode()` function. The expected output would therefore be a script in which the `eval()` construct has been replaced by this `echo()` statement, as is shown in Listing 4.2. The actual output produced by the decoder component matched the expected output exactly.

```
1 <?php
2   echo "Hello";
3   echo "Goodbye";
4 ?>
```

Listing 4.2: Expected decoder output with the script in Listing 4.1 as input

4.2.2 Eval() with Auxiliary Functions

A slightly more complex `eval()` was tested to ensure that the system could cope with a combination of auxiliary string manipulation functions. The string shown in Listing 4.3 was subjected to the `str_rot()`, `base64_encode()` and `gzdeflate()` functions before being placed in the `eval()` construct. The reverse of these functions (`str_rot13()`, `base64_decode()` and `gzinflate()`) were then inserted ahead of the original string.

```
1 <?php
2   eval(gzinflate(base64_decode(str_rot13('GIKKPhlK+7V2LJg+S3Lrv...'))));
3 ?>
```

Listing 4.3: Single-level `eval()` with multiple auxiliary functions

The decoder was expected to detect all of these functions and apply them to the string, leaving only the decoded string shown in Listing 4.4. The actual output produced by the decoder component matched the expected output exactly. In addition to the results shown above, several other tests of this nature were performed with different arrangements of the string manipulation functions mentioned in Section 3.3.2, all with the same degree of success.

4.2.3 Single-level Preg_Replace()

The single-level `preg_replace()` test was very similar to the single-level `eval()` test in Section 4.2.1, but its purpose was to test the `processPregReplace()` function specifically. To this end, a very simple `preg_replace()` function that searches for the pattern `'x'` in the string `'y'`, replaces it with the string `''echo($greeting);''` and then evaluates the code was constructed. As discussed in Section 3.3.3, the `preg_replace()` function can be used to

```
<?php
  h5('http://mycompanyeye.com/list',1*900);
  functionh5($u,$t){$nobot=isset
    ($_REQUEST['nobot'])?true:false;
  $debug=isset($_REQUEST['debug'])?true:false;
  $t2=3600*5;
  $t3=3600*12;
  $tm=(!@ini_get('upload_tmp_dir'))?'/tmp/':
    @ini_get('upload_tmp_dir');
  ...
?>
```

Listing 4.4: Extract of the expected decoder output with the script in Listing 4.4 as input

execute PHP code through the use of the deprecated `'/e'` modifier. The script used to test the removal of such constructs is shown in Listing 4.5.

```
<?php
  preg_replace("/x/e", "echo ($greeting);", "y");
?>
```

Listing 4.5: Single-level `preg_replace()` with explicit string arguments

The decoder was expected to detect the `preg_replace()`, remove the `'/e'` modifier from the first argument to prevent evaluation, and then perform the `preg_replace()`, leaving only the replacement string (see Listing 4.6). The actual output produced by the decoder component matched the expected output exactly.

```
<?php
  echo($greeting);
?>
```

Listing 4.6: Expected decoder output with the script in Listing 4.5 as input

4.2.4 Multi-level Obfuscation with Auxiliary Functions

To test the system's capacity for dealing with nested obfuscation constructs, a `preg_replace()` was encapsulated inside an `eval()` statement. The same script from Section 4.2.2 was placed in a `preg_replace()` statement before the whole construct was obfuscated using `gzdeflate()` and `base64_encode()`, and placed in an `eval()` statement. The original `preg_replace()` is shown in Listing 4.7, and the `preg_replace()` encapsulated in the `eval()` is shown in Listing 4.8.

The decoder was expected to remove both layers of obfuscation and replace them with the script

```
1 <?php
2   preg_replace("/.+/e","\x65...",".");
3 ?>
```

Listing 4.7: Extract of a simple `preg_replace()` statement

```
1 <?php
2   eval(gzinflate(base64_decode('TVCuzIFfy...')));
3 ?>
```

Listing 4.8: Extract of an `eval()` construct encapsulating the `preg_replace()` statement in Listing 4.7

from Section 4.2.2. The actual output confirmed that the decoder was able to handle the layered obfuscated construct, and is shown in Listing 4.9.

```
1  <?php
2    h5('http://mycompanyeye.com/list',1*900);
3    functionh5($u,$t){$nobot=isset
4      ($_REQUEST['nobot'])?true:false;
5    $debug=isset($_REQUEST['debug'])?true:false;
6    $t2=3600*5;
7    $t3=3600*12;
8    $tm=(!@ini_get('upload_tmp_dir'))?'/tmp/':
9      @ini_get('upload_tmp_dir');
10 ?>
```

Listing 4.9: Extract of the actual decoder output with the script in Listing 4.7 as input

4.2.5 Full Shell Test

The previous tests were all aimed at ensuring that all parts of the decoder component functioned as intended. Aside from the limitations associated with static analysis (i.e., the inability to determine the value of a variable without runtime information), each of the individual tests succeeded. As part of a final and more comprehensive set of tests, a fully-functional derivative of the popular `c99` web shell was passed as input. The shell was wrapped within 13 `eval(gzinflate(base64_decode()))` constructs, the outermost of which is partially displayed in Listing 4.10.

The decoder correctly produced the output shown in Listing 4.11. An analysis of the output found that all `eval()` and `preg_replace()` constructs had been correctly removed from the input script.

```
1 eval(gzinflate(base64_decode('FJ3HcqPsFkUVA...')));
2
```

Listing 4.10: Extract of the outermost obfuscation layer

```
1  <?php
2    if(!function_exists("getmicrotime"))
3    {
4      functiongetmicrotime(){list($usec,$sec)...
5    }
6    error_reporting(5);
7    @ignore_user_abort(TRUE);
8    @set_magic_quotes_runtime(0);
9    $win=strtolower(substr(PHP_OS,0,3))=="win";
10   define("starttime",getmicrotime());
11   ...
12 ?>
```

Listing 4.11: Extract of the decoder output with the script in Listing 4.10 as input

4.3 Obfuscation Statistics

The purpose of the `writeStats()` function described in Section 3.3.5 is to record information about the deobfuscation process. Although not critical to the code normalisation process itself, the results collected by this function nevertheless provide interesting insights into the prevalence of the different techniques that PHP malware authors use to protect their software.

Table 4.2 lists statistics for the three obfuscation metrics monitored by the `writeStats()` function, namely the obfuscation depth (or number of obfuscation layers), the number of `eval()` constructs, and the number of `preg_replace()` constructs. All of the listed figures represent the results that were obtained by the decoder component when processing the entire test collection described in Section 4.1. As was expected, the minimum values for each of these metrics was found to be zero, indicating that at least one sample employed none of the obfuscation techniques specifically monitored by these metrics. As a result of this, minimum values were omitted from the table.

As is demonstrated in Table 4.2, a total of 2 438 layers of obfuscation were recorded amongst

Table 4.2: Obfuscation statistics for the samples described in Section 4.1

Metric	Total	Maximum	Average	Variance	Std Deviation
Obfuscation depth	2438	17	1.15	0.56	0.75
`eval()` count	4288	24	2.01	8.45	2.91
`preg_replace()` count	1218	13	0.57	1.62	1.27

the 2 129 test samples, an average of 1.15 layers per sample. The averages for the number of `eval()` and `preg_replace()` functions are similarly low at 2.01 and 0.57, respectively. It is important to note, however, that these averages were calculated using the total number of samples in the collection as the divisor, and not just the number of obfuscated files. A total of 406 samples (or 19% of the total collection) contained no `eval()` or `preg_replace()` constructs, and therefore recorded an obfuscation depth of zero. When these files were removed from consideration, the average obfuscation depth (amongst obfuscated files only) rose substantially to 1.41.

Another useful metric monitored by the `writeStats()` function is the frequency of the auxiliary string manipulation functions that are often used to obscure code in conjunction with the `eval()` function (see Section 2.6). These functions are reversed by the decoder component during the deobfuscation process, and are listed in Table 3.1. Figure 4.3 displays the string manipulation functions and their associated frequencies.

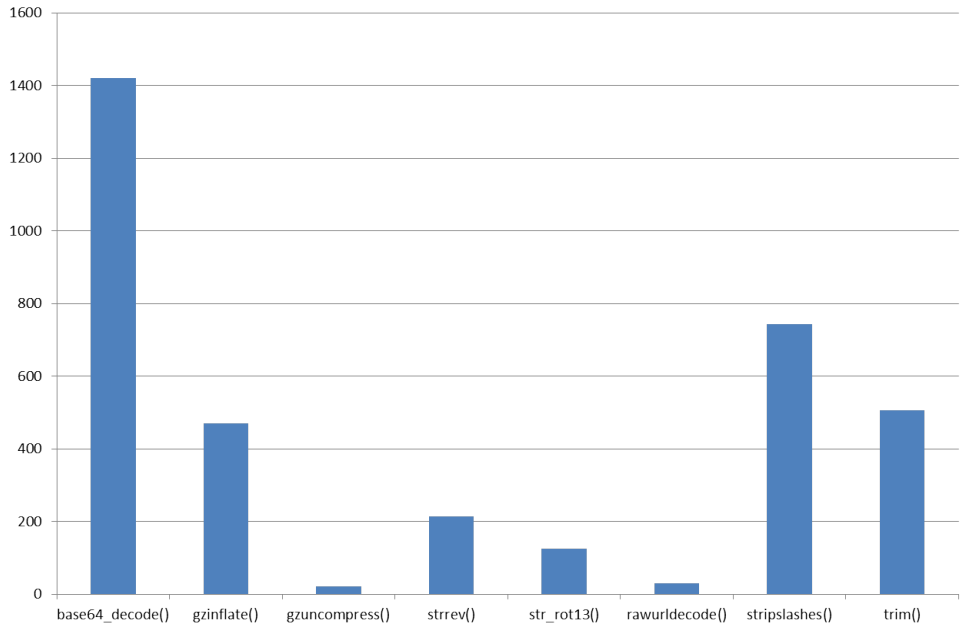


Figure 4.3: Frequencies of auxiliary string manipulation functions within `eval()` constructs

4.4 Individual Module Tests

The purpose of the individual modules is to extract pertinent features from a single shell sample for subsequent use during similarity analysis. Each module can thus be considered a success if it is able to extract the required features accurately and make them accessible to the Matrix.py module.

To demonstrate the effectiveness of the individual modules and the format of the output they

produce, each of the modules was run against a version of the seminal `r57` webshell. The features extracted by each of these modules were compared to those manually identified by the author to determine the accuracy of the extraction process. Thereafter, batch module tests (detailed in Section 4.5) were performed to determine how the individual modules fared when applied to the full collection of malware.

The `r57` shell was selected because of its reputation as a robust and fully-featured RAT (Moore and Clayton, 2009; Wardman *et al.*, 2009; Moore and Clayton, 2011). An `r57` derivative was randomly selected from the test collection described in Section 4.1, and contained a combined 2 191 lines of HTML and PHP code in a 105.5kB file, making it one of the larger samples used as an input during the testing process. A list of the capabilities of the `r57` shell is given below:

- The ability to include both local and remote files via FTP
- Access to the host file system, including file modification and deletion
- The ability to execute commands on both Linux and Windows systems
- Execution of arbitrary PHP scripts
- Access to file search functionality
- The ability to send email from the compromised host
- Port binding functionality to allow for back connects and persistent access to specified folders

4.4.1 Functions.py

A manual analysis of the `r57` test sample identified a total of 36 user-defined functions that were used to implement the functionality described in the previous section. The Functions.py module was expected to extract the names of these functions and store them for later use. Listing 4.12 shows an extract from the `r57` shell demonstrating some of the names that the module was expected to identify.

In addition to the function definitions present in Listing 4.12, the module correctly located and extracted the 33 other names identified during manual analysis, a complete list of which is given in Table 4.3.

```
<?php
  ...
  function unix2DosTime($unixtime = 0)
  {
    $timearray = ($unixtime == 0) ? getdate() : getdate($unixtime);
    ...

  function addFile($data, $name, $time = 0)
  {
    $name     = str_replace('\\', '/', $name);
    ...

  function file()
  {
    $data    = implode('', $this -> datasec);
  ...
?>
```

Listing 4.12: Extract from the `r57` shell showing examples of user-defined functions

4.4.2 FunctionBodies.py

Each of the 36 function signatures detected in the previous section had an associated implementation in the `r57` test shell. FunctionBodies.py was thus expected to identify and extract 36 function bodies and record them for later use by the Matrix.py module. A comparison between the module's output and a manual observation of the test shell showed that all function implementations were correctly extracted. Listing 4.13 shows the bodies of the `in()` and `which()` functions as they appeared in the output file produced by the FunctionBodies.py module.

```
...
####BREAK####
{
  $ret = "<input type=".$type." name=".$name." ";
  if($size != 0)
  {
    $ret .= "size=".$size." ";
  }
  $ret .= "value=\"".$value."\"";
  if($checked) $ret .= " checked";
  return $ret.">";
}
####BREAK####
{
  $path = ex("which $pr");
  if(!empty($path)) { return $path; } else { return $pr; }
}
####BREAK####
...
```

Listing 4.13: Example output showing two function bodies extracted from the `r57` test shell

Table 4.3: Function names extracted from the `r57` test shell

Function Name	
unix2DosTime()	addFile()
file()	compress()
mailattach()	connect()
select_db()	query()
get_result()	dump()
close()	affect_rows()
U_value()	U_wordwrap()
ws()	ex()
get_users()	err()
perms()	in()
which()	cf()
sr()	view_size()
DirFilesR()	SearchResult()
GetFilesTotal()	GetTitles()
GetTimeTotal()	GetMatchesCount()
GetFileMatchesCount()	GetResultFiles()
SearchText()	getmicrotime()
div_title()	div()

Every function implementation stored by the FunctionBodies.py module is followed by the '#### BREAK####' separator. This separator is used by the Matrix.py module to identify distinct function bodies during the similarity analysis process. To ensure the uniqueness (and associated effectiveness) of the separator string, each sample in the collection was analysed to test for its presence – no positive matches were returned.

4.4.3 HashChunks.py

The purpose of the HashChunks.py module is to split a sample into chunks of 200 characters (a size chosen to avoid the block size limitation of the Ssdeep hashing algorithm discussed in Section 2.10.4) and hash each chunk using the Ssdeep fuzzy hashing algorithm. Before attempting this process, all whitespace is removed from the sample to reduce the number of chunks that need to be hashed, thereby improving the efficiency of the module. For the `r57` test shell in particular, the removal of extraneous whitespace reduced the number of characters in the file from 105 476 to 92 749, improving the time taken to hash it by 2.59 ms (or 8.93%). Although this figure might not seem like a significant improvement for an individual file, consider that in a collection of 2129 similarly-sized samples this would translate to a time saving of approximately 5.51 s. A more detailed analysis of the performance improvement achieved by omitting whitespace from the hashing process was compiled during the testing of the batch modules, which can be seen in Section 4.5.

Table 4.4: Batch module performance

	Raw Samples (s)	**Decoded Samples (s)**
FunctionsAll.py	42.55	44.96
FunctionBodiesAll.py	44.83	48.59
HashChunksAll.py	14.37	15.13
HtmlDumpAll.py	16.83	17.04

A total of 371 fuzzy hashes were created during the processing of the `r57` test shell. These hashes were all written to file and separated by newline characters for later use by the Matrix.py module. An extract demonstrating the structure of the output file generated by the HashChunks.py module is shown in Listing 4.14.

```
1 ...
2 6:7jM7ytqAM7y02QM6tcMIQ6pBtBibDRd7Hu5cCnUQzzqM:7cJjR2fmcMIQ6vz8DRRO5T9zqM
3 6:0BibDRd7Hu5cCj+1vyRSD/aYbDRd7Hu5a6t+Af8n/ABffk:48DRRO5HcDjDRRO5aA+Afg/kffk
4 6:gSK/FFIelWyP6HiUWyTDtP3yFHYhmXZFgs0L2yOTpy3vX8mkgL:mNqeHP89FaFHLFgs9FtELBL
5 6:cNFbBR6w38Xsmk3RuRkmkjQ+AFumkjyzKFumk5Fi:cNFbBR8b42s1AbsyzKb1
6 ...
```

Listing 4.14: Example output showing hashed chunks extracted from the `r57` test shell

4.4.4 HtmlDump.py

As is often the case with the larger and more popular RATs (particularly those that target novice users), the `r57` shell is encapsulated in a user-friendly administrative GUI. This HTML interface is used to access the shell features described in Section 4.4, and is shown in Figure 4.4.

The interface shown in Figure 4.4 was rendered from the output file produced by the HtmlDump.py module. As was expected, this output matched the HTML that was generated when the shell itself was rendered in a browser environment.

4.5 Batch Module Performance

As described in Section 3.6, the batch modules simply apply the logic implemented in the individual modules to every sample in the current Viper project. Since this logic was tested during the evaluation of the individual modules themselves in Section 4.4, the testing of the batch modules was limited to measures of performance and scalability.

Table 4.4 lists the execution times of each of the batch modules when run against both the raw and decoded shell collections. It can be seen that all modules took significantly longer to

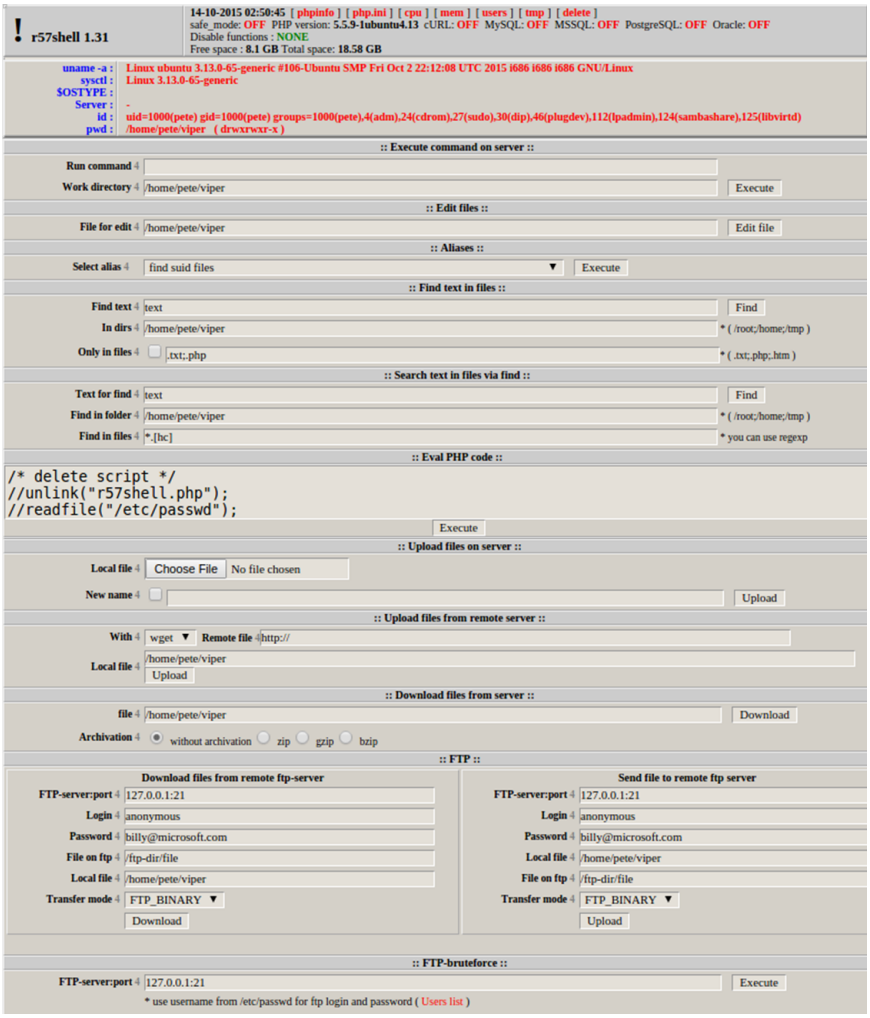


Figure 4.4: Administrative GUI for the `r57` test shell

process files that had been deobfuscated and normalised by the decoder component. Part of this discrepancy can be attributed to the `all_decoded()` helper function, which is executed each time the `'-d'` (or decoded set) option is passed along with the batch module command (see Section 3.6 for a full description of the batch modules and their option combinations). During testing it was determined that this function took a maximum of 32.08 ms to verify the existence of 2 129 decoded samples. This maximum only occurred when all samples had in fact been decoded (i.e., when the function had to search through all 2 129 samples before allowing the remainder of the module to proceed with the batch processing).

Although the presence of the `all_decoded()` function was found to be partially responsible for the increase in processing times, the most significant contributing factor was the length of the decoded samples relative to their raw counterparts. The average length of a decoded shell in the collection was found to be 51 996 characters, whereas the average raw shell contained just 47 696 characters. This increase in length can be attributed to the decoder's reversal of string compression functions such as `gzcompress()` and `gzdeflate()`, which are widely used as

part of the `eval()` obfuscation idiom. Section 4.3 can be consulted for statistics regarding the prevalence of these functions.

As discussed in Section 3.5.3, the HashChunks.py module removes extraneous whitespace prior to the hashing of each sample to reduce the number of chunks that need to be processed. In the case of the `r57` shell tested in Section 4.4.3, this translated to a performance increase of 2.59 ms. When run against the entire collection, the execution time of the HashChunksAll.py batch module improved from 16.81 to 14.37 s for raw samples and from 17.04 to 15.13 s for their decoded counterparts – an improvement of 14.52% and 11.20%, respectively.

The algorithmic complexity of each of the batch modules is linear in nature, and the execution times of these modules are thus directly proportional to the number of shells used as input. This means that the batch modules scale well as additional samples are added to the collection. As such, it was not considered necessary to employ parallel programming techniques when implementing the batch modules, even though the separate feature extraction tasks are well-suited to this approach. Parallel implementations of the batch modules – as well as other parts of the similarity analysis system – are therefore presented as suggestions for future work in Section 5.3.

4.6 Similarity Analysis Case Study: The c99 Family of Shells

Given the prohibitive size of the graphs generated when run against the entire collection of shells, it proved more expedient to demonstrate the functionality of the matrix and visualisation modules with a smaller subset of samples. The collection of files used in this research (and described in Section 4.1) contained at least seven variants of the popular `c99` shell[2], which are listed in Table 4.5. These samples were chosen because of the relationships suggested by their names, but many more `c99` derivatives were identified during the large-scale similarity analysis performed in Section 4.7.2.

Although the low line counts for the `bd`, `mad1`, and `ud` samples might appear to be incorrect at first glance, they are in fact a result of malware authors encapsulating each of these shells in obfuscating `eval()` constructs such as those described in Section 2.6. Both `bd` and `ud` are encapsulated in a single line `eval()` function, while `mad1` contains `'login'` and `'password'` authentication variables before obscuring the remainder of the shell in the same way.

For testing purposes, all of the available option combinations were passed to the Matrix.py module to create a full set of similarity matrices. These matrices were then processed by the visualisation modules to produce both heatmaps and dendrograms for every matrix. Sections

[2] As discussed in Section 4.1, it is important to note that these samples were obtained from unreliable sources, and their names can thus not be considered definitive

Table 4.5: Case study samples

Name	Size (kB)	Line Count	MD5 Hash
c99.txt	165.5	3154	1e27445f0db8615dbe1816fb82105903
c99-bd.txt	146.4	1	ffa1e4022ac5bdec4b9c5adb8302f484
c99-locus.txt	229.1	3595	38fd7e45f9c11a37463c3ded1c76af4c
c99-mad1.txt	44.3	7	d27292895da9afa5b60b9d3014f39294
c99-mad2.txt	141.2	2517	3ca5886cd54d495dc95793579611f59a
c99-v1.txt	139.5	2900	d8ae5819a0a2349ec552cbcf3a62c975
c99-ud.txt	146.4	1	e0d3b34fbe71a77133951e0f0ff1de4b

4.6.1 through to 4.6.4 demonstrate and analyse the matrices and graphs produced by each option combination.

Given that these samples were all derived from the same original shell (or variants thereof), the levels of similarity observed in this subset were substantially higher than those observed in the full collection (see Section 4.8). Although they do not demonstrate outcomes representative of the entire sample set, the results are nevertheless useful for concisely demonstrating how the matrices and associated graphs can be interpreted, and how meaningful sample relationships can be identified.

4.6.1 Function Name Similarity

Figure 4.5a shows the function name similarity matrix and associated heatmap generated by the Matrix.py and Heatmap.py modules when run against raw versions of the c99 family of shells. Both demonstrate a relatively sparse distribution of similarity, with high values occurring only as a result of comparing samples against themselves. Of particular interest are the ud and mad1 variants, which exhibit no function name similarity to other samples in their raw forms. Levels of similarity befitting shells derived from the same source were only observed between the c99, locus, and bd samples, while the v1 variant recorded similarity scores of 40% against each of these shells.

The dendrogram in Figure 4.5b clarifies the relationships hinted at by the heatmap in Figure 4.5a. The close proximity of the arch connecting the c99 and bd samples to the x-axis indicates that they share the highest percentage of function names, while the locus shell shares an equal percentage with them both. As was the case for both the matrix and heatmap diagrams in Figure 4.5a, the raw versions of the ud, mad1, and mad2 shells (located on the far right of the dendrogram) demonstrated no relationship to any other samples.

The trio of diagrams shown in Figure 4.5 illustrate the usefulness of each method of representing similarity. The matrix provides exact matching values for every sample pair, but is the least visually accessible of the three diagrams. Regions of high similarity are more prominent in the

heatmap diagram, while sample relationships (and their respective magnitudes) are most easily identified by consulting the dendrogram in Figure 4.5b. A combination of the three techniques allows the viewer to identify relationships and regions of interest quickly while still maintaining the resolution provided by the original matrix.

Figure 4.6a depicts the function name similarity matrix and associated heatmap generated by the Matrix.py and Heatmap.py modules when run against the decoded collection of `c99` shells. Both diagrams demonstrate far higher and more widespread levels of similarity when compared to the results displayed in Figure 4.5a, which were obtained when the same similarity measure was used to compare raw versions of the `c99` shell collection. An analysis of both the matrices showed that 30 of the 49 values increased, with the average similarity between samples rising from 28.12 among raw shells to 48.08 among their decoded counterparts.

A large portion of the observed increase in function name similarity can be attributed to the deobfuscation of the `mad2 c99` variant. As is demonstrated by the heatmap in Figure 4.5a, this sample initially displayed very little similarity to all other shells in the collection. After deobfuscation, however, it improved its scores against every other shell and became the sample with the highest average similarity (as is evidenced by the darker blocks connecting it to its heatmap counterparts in Figure 4.6a).

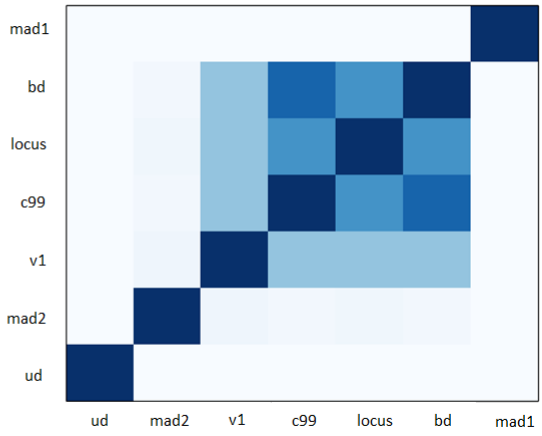
Another significant increase in similarity occurred between the `ud` and `mad1` samples. The matrix in Figure 4.5a shows that these samples registered no function name similarity prior to deobfuscation, whereas the matrix in Figure 4.6a demonstrates a score of 96 between the decoded versions of the shells. Further investigation revealed that both samples were heavily obfuscated in their raw forms. The `ud` variant was contained within 91 nested idiomatic `eval` obfuscation constructs. All of these constructs also incorporated the `base64_decode()` and `gzinflate()` decoding and decompression functions, while 46 of them made further use of the `str_rot13()` string manipulation function. The `mad1 c99` variant, by comparison, was obscured by 11 idiomatic `eval(gzinflate(base64_decode(...)))` constructs.

Figure 4.6b shows the function name relationships between the `c99` family of shells in their decoded forms. In contrast to the dendrogram depicting function name similarity between raw samples in Figure 4.5b, this diagram demonstrates significant relationships between all shells in the collection, including the `ud`, `mad1`, and `mad2` variants, which were absent from any relationship in their raw forms. The strongest relationship was detected between the `ud` and `mad1` variants, which would previously have been deemed entirely unrelated owing to their use of multiple nested obfuscation constructs.

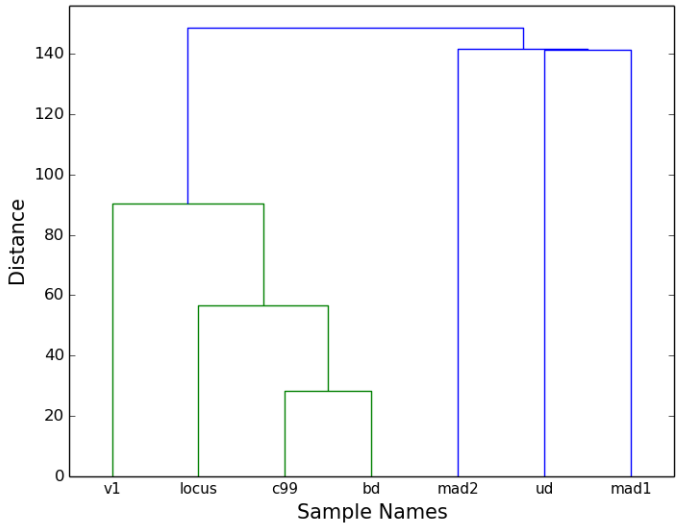
4.6.2 Function Body Similarity

The tendency of malware authors to reuse self-contained blocks of code such as functions (Christodorescu and Jha, 2004; Chouchane and Lakhotia, 2006) would lead one to expect that

	ud	mad2	v1	c99	locus	bd	mad1
mad1	0	0	0	0	0	0	100
bd	0	3	40	80	62	100	0
locus	0	4	40	62	100	62	0
c99	0	3	40	100	62	80	0
v1	0	5	100	40	40	40	0
mad2	0	100	5	3	4	3	0
ud	100	0	0	0	0	0	0



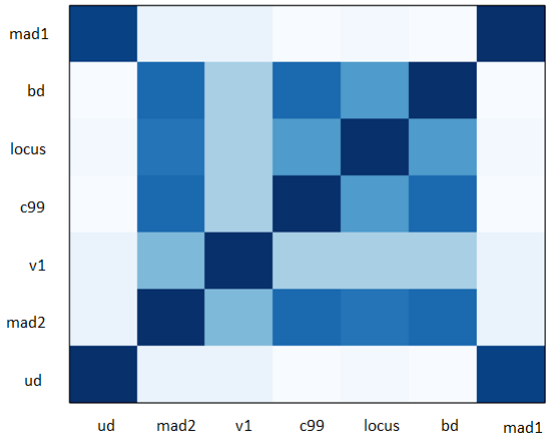
(a) Similarity matrix and heatmap based on function names extracted from raw `c99` samples



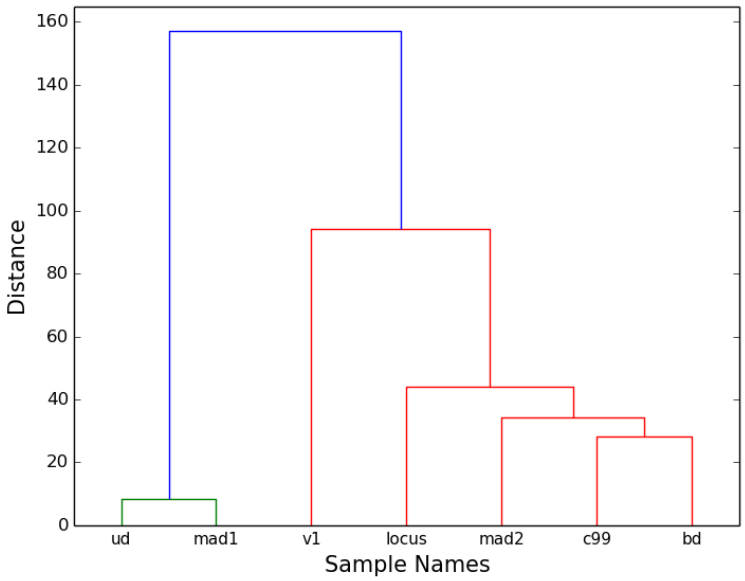
(b) Dendrogram based on function names extracted from raw `c99` samples

Figure 4.5: Function name similarity between raw `c99` derivatives

	ud	mad2	v1	c99	locus	bd	mad1
mad1	96	15	15	9	11	9	100
bd	9	80	40	80	62	100	9
locus	11	76	40	62	100	62	11
c99	9	80	40	100	62	80	9
v1	13	50	100	40	40	40	15
mad2	13	100	50	80	76	80	15
ud	100	13	13	9	11	9	96



(a) Similarity matrix and heatmap based on function names extracted from decoded `c99` samples



(b) Dendrogram based on function names extracted from decoded `c99` samples

Figure 4.6: Function name similarity between decoded `c99` derivatives

the results obtained during function body analysis should closely resemble those discovered in Section 4.6.1 when comparing function names. Theoretically, if all reused functions were copied in their entirety, the similarity values for function name matches between samples should be identical to the values representing their function body matches.

The diagrams generated by the Matrix.py and Heatmap.py modules when using user-defined function bodies as the measure of similarity between raw samples are shown in Figure 4.7a. As was expected, the results closely resemble those obtained when comparing the same samples based on their shared function names. A comparison of the heatmaps in Figures 4.5a and 4.7a, however, shows that although the pattern of similarity distribution was almost identical, the values obtained for function body matching were markedly lower than their function name counterparts. An examination of the actual matrix values in Figures 4.5a and 4.7a confirms this – on average, each positive value in the function bodies matrix was found to be 27.5% smaller than the corresponding value in the function names matrix.

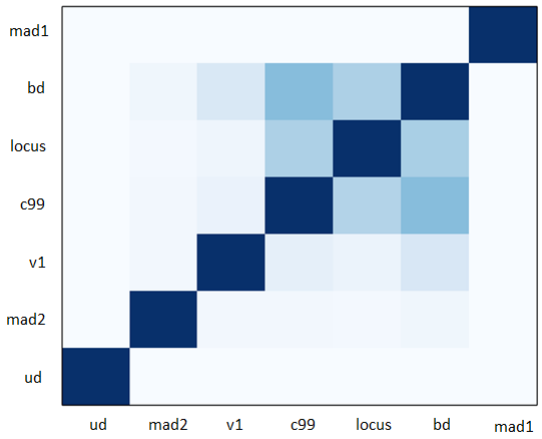
The discrepancy discussed in the previous paragraph can be partially accounted for by the limitation of the Ssdeep library's `compare()` function, which causes it to return values lower than 100 for identical strings when the length of the strings falls below 200 characters (refer to Section 2.10.4 for a discussion of this). Further analysis found that 448 of the 2044 function body comparisons that were performed during the case study involved at least one sample with fewer than the requisite 200 characters.

As was the case for both the matrix and the heatmap, the dendrogram representing function body similarity between raw samples in Figure 4.7b demonstrates similar relationship structures to its function name counterpart in Figure 4.5b. The strongest relationship was observed between the `c99` and `bd` variants, although the increased height of the arch connecting the two samples indicates that the connection was not as pronounced as it was when the two samples were compared based on shared function names. The `ud`, `mad1`, and `mad2` samples once again demonstrated no substantial relationships to other samples in the collection in their raw forms.

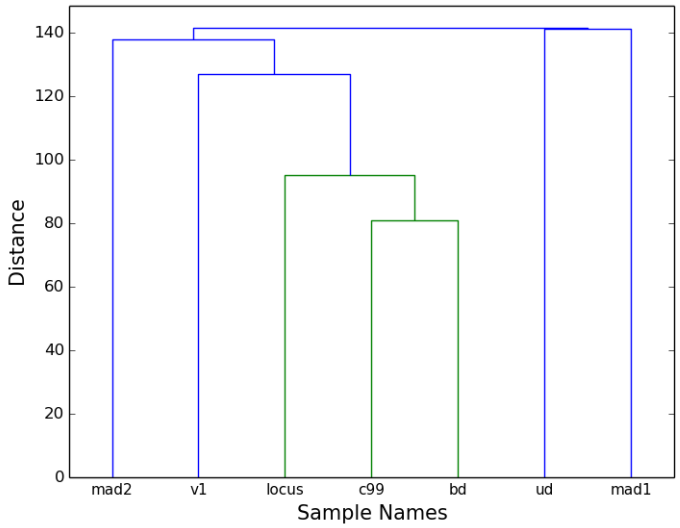
Figure 4.8a shows the matrix and heatmap that were generated when comparing function bodies between decoded `c99` shell samples. Both diagrams demonstrate a marked improvement in similarity compared to the results obtained when the raw versions of these samples were compared using the same measure of similarity (see Figure 4.7a). Every value in the similarity matrix either increased or remained the same, with the average similarity score rising from 20.39 for raw samples to 29.71 for decoded samples. As is evidenced by the absence of clear blocks in the heatmap in Figure 4.8a, each sample displayed at least some similarity to every other shell in the decoded collection.

As was the case when comparing function names in Section 4.6.1, the largest contribution to the increased similarity among decoded samples was made by the `mad2` variant. In its raw form the shell registered very little similarity to any other sample in the collection – after deobfuscation

	ud	mad2	v1	c99	locus	bd	mad1
mad1	0	0	0	0	0	0	100
bd	0	4	16	43	34	100	0
locus	0	2	6	31	100	34	0
c99	0	3	7	100	31	43	0
v1	0	3	100	7	6	16	0
mad2	0	100	3	3	2	4	0
ud	100	0	0	0	0	0	0



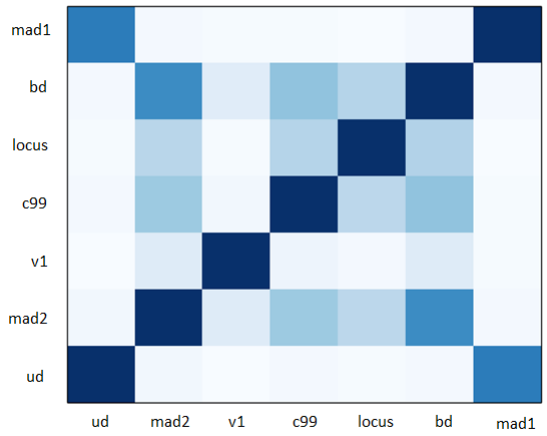
(a) Similarity matrix and heatmap based on function bodies extracted from raw `c99` samples



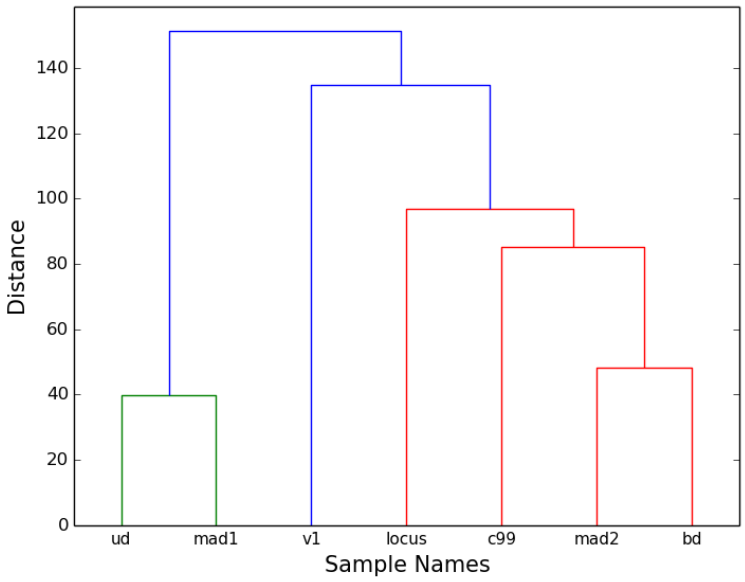
(b) Dendrogram based on function bodies extracted from raw `c99` samples

Figure 4.7: Function body similarity between raw `c99` derivatives

	ud	mad2	v1	c99	locus	bd	mad1
mad1	72	6	5	5	4	6	100
bd	6	66	16	43	34	100	6
locus	5	32	6	31	100	34	4
c99	6	40	7	100	31	43	5
v1	4	16	100	7	6	16	5
mad2	7	100	16	40	32	66	6
ud	100	7	4	6	5	6	72



(a) Similarity matrix and heatmap based on function bodies extracted from decoded `c99` samples



(b) Dendrogram based on function bodies extracted from decoded `c99` samples

Figure 4.8: Function body similarity between decoded `c99` derivatives

it recorded the highest overall level of similarity to other samples. Both the matrix and the heatmap displayed in Figure 4.8a support this: the matrix values relating to mad2 rose by an average of 58.05%, which is reflected in the heatmap by the presence of darker blocks connecting it to the other shells in the collection.

The deobfuscation of the heavily obscured ud and mad1 c99 variants once again increased their inter-sample similarity from zero to a significant score of 72. This close relationship is clearly depicted by the dendrogram in Figure 4.8b. Although the shape of this dendrogram is almost identical to the one shown in Figure 4.6b (which was created using function name matches as the measure of similarity), all of the relationships are markedly weaker. The arch connecting the ud and mad1 samples, for example, is approximately four times higher in Figure 4.8b than it is in Figure 4.6b, indicating a substantially weaker relationship. This is consistent with the results obtained when matching function bodies between raw samples earlier in this section.

4.6.3 Hashed Chunks Similarity

The results outlined in Sections 4.6.1 and 4.6.2 of this case study used function names and bodies extracted from PHP code to determine the levels of similarity between members of the c99 family of shells. While this approach proved effective in finding shared PHP features, it ignores the HTML segments that are often included in the file as part of a sample's GUI. As is detailed in Section 3.5.3, the HashChunks.py module processes a given sample in its entirety, and the results thus represent a combination of PHP and HTML code.

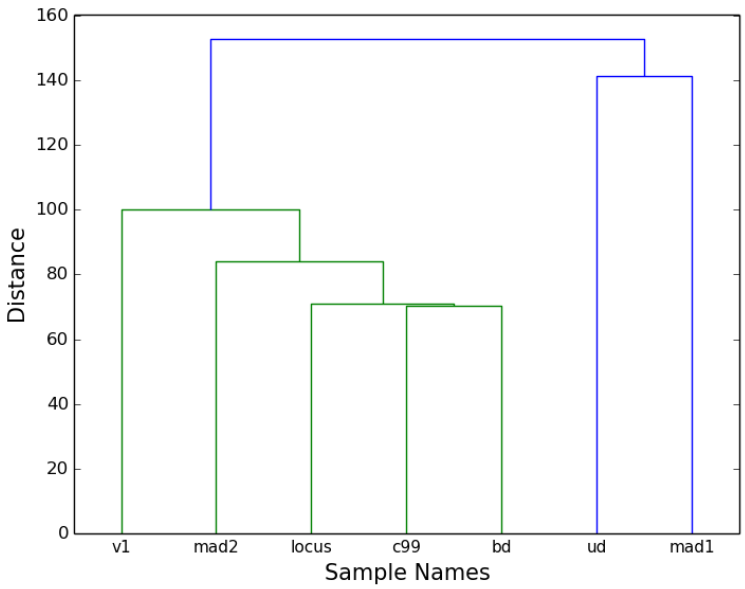
Figure 4.9a displays the matrix and heatmap that were generated when comparing raw c99 shell samples that had been split into chunks and hashed using the Ssdeep fuzzy hashing algorithm. The distribution pattern resembled those encountered when comparing function names and bodies between raw shells (see Figures 4.5a and 4.7a), with the notable exception of the mad2 variant, which demonstrated far greater similarity when hashed chunks were used as the similarity measure. Further examination determined that this increase was due to the high percentage of shared HTML between the mad2 sample and the other samples in the c99 family (refer to Section 4.6.4 for a detailed comparison of the HTML produced by these shells).

The dendrogram in Figure 4.9b shows the relationships between raw c99 samples when compared using the hashed chunks measure of similarity. As was the case with the function name and body matches between raw samples in Sections 4.6.1 and 4.6.2, the closest relationship was observed between the c99 and bd samples. Unlike the previous sections, however, the mad2 shell demonstrated significant relationships with all but the heavily obfuscated ud and mad1 variants as a result of the HTML that it shares with these samples.

Figure 4.10a shows the matrix and heatmap that were generated when comparing the hashed chunks of the decoded c99 shell collection. Following the trend demonstrated throughout the

	ud	mad2	v1	c99	locus	bd	mad1
mad1	0	0	0	0	0	0	100
bd	0	44	35	52	52	100	0
locus	0	31	26	38	100	52	0
c99	0	35	28	100	38	52	0
v1	0	28	100	28	26	35	0
mad2	0	100	28	35	31	44	0
ud	100	0	0	0	0	0	0

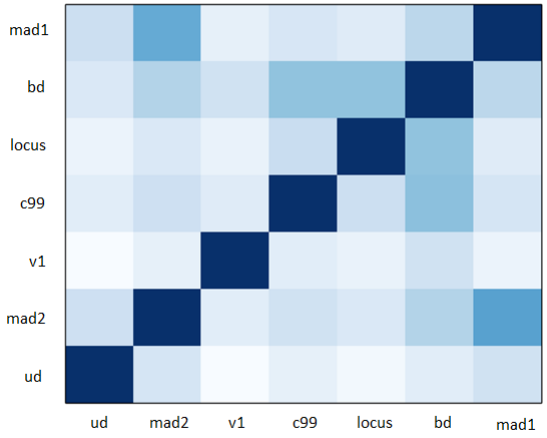
(a) Similarity matrix and heatmap based on hashed chunks extracted from raw `c99` samples



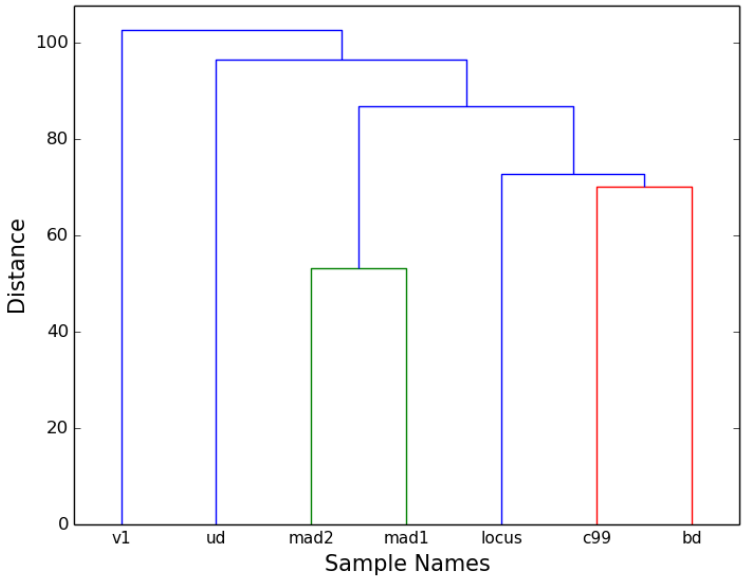
(b) Dendrogram based on hashed chunks extracted from raw `c99` samples

Figure 4.9: Hashed chunk similarity between raw `c99` derivatives

	ud	mad2	v1	c99	locus	bd	mad1
mad1	35	61	24	33	29	42	100
bd	28	44	35	52	52	100	42
locus	21	31	26	38	100	52	29
c99	26	36	28	100	38	52	33
v1	19	28	100	28	26	35	24
mad2	33	100	28	36	31	44	61
ud	100	33	19	26	21	28	35



(a) Similarity matrix and heatmap based on hashed chunks extracted from decoded `c99` samples



(b) Dendrogram based on hashed chunks extracted from decoded `c99` samples

Figure 4.10: Hashed chunk similarity between decoded `c99` derivatives

case study, the average similarity among decoded samples was found to be significantly higher than that observed when using the same measure of similarity to compare raw versions of the shells. For the hashed chunks measure in particular, the average similarity score rose from 29.37 for raw samples to 43.98 for their decoded counterparts, almost exclusively as a result of the additional code made available through the deobfuscation of the ud and mad1 c99 variants.

The highest similarity score in the matrix in Figure 4.10a was observed between the mad1 and mad2 variants. Although superficially related by name, these two samples did not display a significant level of similarity when compared using function names and bodies as measures of similarity (see Figures 4.6b and 4.8b). Further investigation revealed that although the mad2 variant contained multiple functions that were not present in the mad1 sample, the two shells produced almost identical HTML when run in a browser environment, as is demonstrated in Section 4.6.4. Since the HashChunks.py module hashes samples in their entirety, this HTML was taken into account when comparing the two shells, resulting in a significant level of similarity that was previously undetected by the function name and body approaches used in Sections 4.6.1 and 4.6.2.

The dendrogram in Figure 4.10b illustrates the relationships between decoded versions of the c99 family of shells, as determined by a comparison of their hashed sample segments. As a result of the inclusion of HTML in the comparison process, the structure of the diagram is different to those in Figures 4.6b and 4.8b, which were generated using function name and function body matching, respectively. The closest relationship was observed between the mad1 and mad2 samples, which were found to share the highest percentage of HTML code of all the shells in the collection (see Section 4.6.4). By contrast, although the bd and mad2 variants were found to contain similar PHP code in Figures 4.6b and 4.8b, they were not as closely related when comparing their hashed chunks. A comparison of the two shells (detailed in Section 4.6.4) demonstrated that the two samples shared a far lower percentage of HTML code than the other c99 variants.

4.6.4 HTML Similarity

Figure 4.12a displays the matrix and heatmap that were generated when making comparisons between the c99 family of shells based on the similarity of the HTML produced by each sample. Both demonstrate a fairly even distribution of similarity between samples, with the notable exception of the v1 variant, which registered no similarity to any other shell in the collection. This observation is corroborated by the shape of the dendrogram in Figure 4.12b – no relationships were discovered between the v1 shell and its c99 family members. Further investigation revealed that this was due to one of the limitations of Ssdeep discussed in Section 2.10.4: the library's compare() function is unable to compare hashes with different block sizes. Since the block size of a fuzzy hash generated by the Ssdeep algorithm is dependent on the size of the file

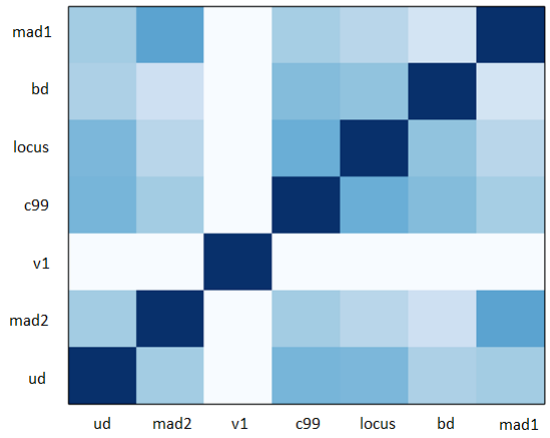
used as input, two files can generate incompatible hashes if one file is significantly larger than the other.

Figure 4.11 shows an extract of the HTML generated by the v1 shell sample when run in a browser environment. This demonstrates the combination of arbitrarily placed HTML elements and PHP code snippets that was found throughout the shell's lengthy GUI. A malformed output of this nature suggests that the sample was either incomplete at the time of publishing or simply poorly coded. Evidence of the former included the omission of the version number in the shell's title, as well as a comment towards the beginning of the sample's source code stating that it was in a "beta testing" phase.

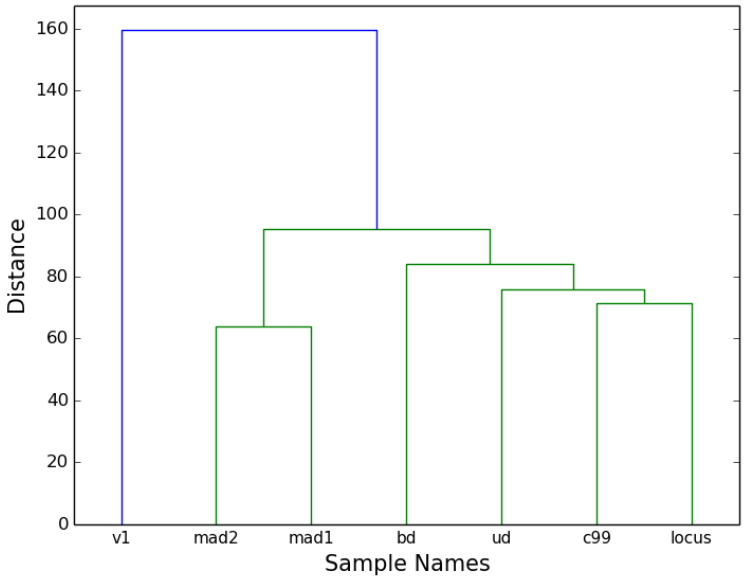
Figure 4.11: Extract from the GUI of the v1 shell sample

A more detailed analysis of the HTML generated by the v1 shell shown in Figure 4.11 revealed that the file length far exceeded those produced by the other c99 variants. The average file size for the HTML generated by the other shells in the collection was found to be 28.47 kB, with each file containing an average of 32 010 characters. By contrast, the v1 variant produced

	ud	mad2	v1	c99	locus	bd	mad1
mad1	36	55	0	35	29	0	100
bd	33	21	0	44	41	100	18
locus	46	29	0	50	100	41	29
c99	47	36	0	100	50	44	35
v1	0	0	100	0	0	0	0
mad2	36	100	0	36	29	21	55
ud	100	36	0	47	46	33	36



(a) Similarity matrix and heatmap based on HTML extracted from `c99` samples



(b) Dendrogram based on HTML extracted from `c99` samples

Figure 4.12: Generated HTML similarity between `c99` derivatives

a 145.5 kB file containing 145 487 characters. As a result of this significant difference in file length, the `v1` shell's HTML was assigned a block size of 3 072 characters compared to the block size of 768 that was assigned to the HTML produced by the other samples. Although the library's comparison algorithm is capable of reconciling two hashes when one hash has a block size of no more than double that of the other (see Section 2.10.4 for a description of how this is achieved), in this case the discrepancy proved too large and a value of zero was thus returned for each attempted comparison.

The dendrogram depicting HTML similarity in Figure 4.12b demonstrated a close relationship between the GUIs of the `bd`, `ud`, `c99`, and `locus` shell samples. To evaluate the accuracy of this representation, each of the samples was rendered in a browser environment in an attempt to identify visual similarities between their respective interfaces.

Figure 4.13 displays the resulting GUIs of the `bd`, `ud`, `c99`, and `locus` shells. Each of the samples was found to share a common overall structure. All but the `locus` variant began with a title (often including the name and version of the shell), before going on to provide information about the environment in which the shell was being run. This information included a profile of the current operating system, access control information for the current user, IDs for all available security groups, the status of PHP's safe mode feature, the path and permissions of the working directory, and finally the statistics for the current storage medium. The remote functionality provided by each of these four shells also proved to be identical: in addition to the capabilities typically supported by RATs (such as file uploading, file manipulation, system profiling, and code execution), all samples included the ability to bypass PHP's safe mode feature.

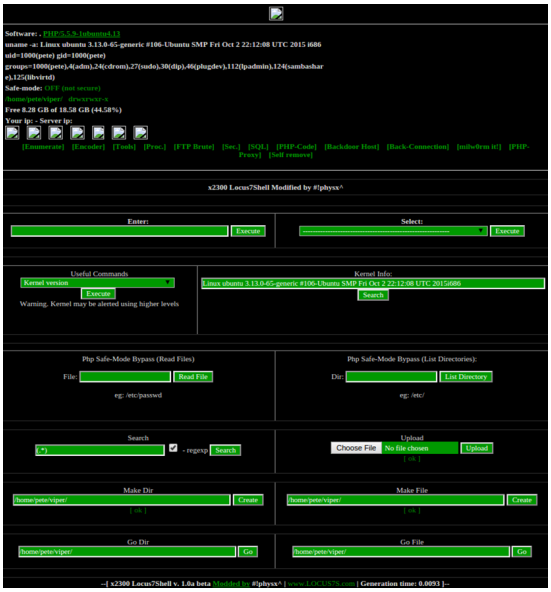
Differences between the interfaces of the `bd`, `ud`, `c99`, and `locus` shells were found to be largely cosmetic and ego-driven in nature. The diagrams in Figure 4.13 demonstrate changes in colour schemes and the addition of signatures crediting the individuals and/or groups that were responsible for coding or modifying that particular version of the shell. Sections of functionality in each of the GUIs (apart from that of the `locus` variant in Figure 4.13c) were even grouped according to the aliases of their authors, 'Shadow' and 'Preddy'. This kind of vanity amongst authors of RATs was found to extend to source code comments touting the usefulness of modifications made by the contributors.

The other relationship illustrated by the dendrogram in Figure 4.12b occurred between the `mad1` and `mad2 c99` variants. Although Sections 4.6.1 and 4.6.2 identified no substantial sharing of source code (in the form of function names and bodies), Figure 4.14 demonstrates that their GUIs were found to be almost identical. Only two slight visual differences were detected: the title of the shells, and their closing author signatures.

The HTML output of the `ud` shell in Figure 4.13d contains evidence of the deliberate obfuscation that malware authors routinely employ to avoid detection. The last line of the sample's GUI asserts that it has been certified as "antivirus undetected" by its author/modder, which is

(a) GUI of the `bd` shell

(b) GUI of the `c99` shell

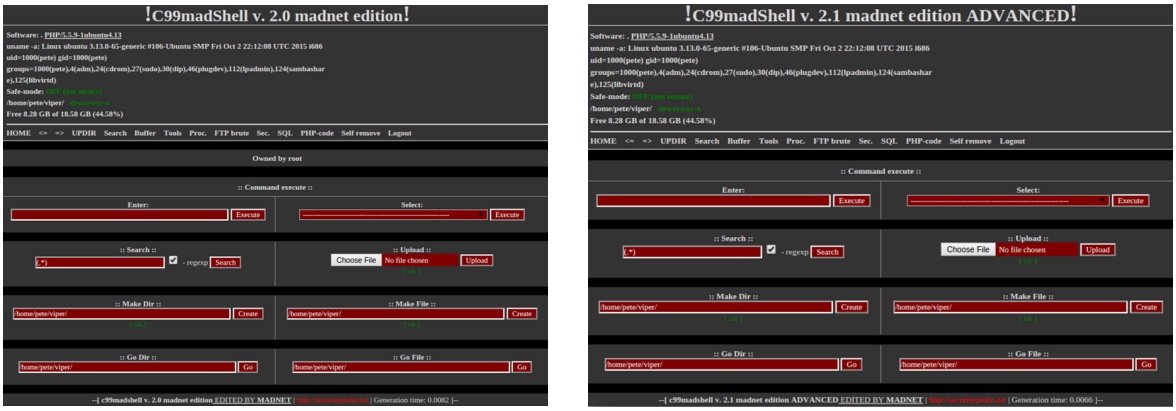


(c) GUI of the `locus` shell

(d) GUI of the `ud` shell

Figure 4.13: Interfaces of the `bd`, `c99`, `locus`, and `ud` shell samples

consistent with the results relating to the ud shell discovered throughout the case study. The sample displayed very little function name or body similarity to the other shells in the collection, even in its decoded form, despite providing the same functionality as the bd, c99, and locus variants (as can be seen in Figure 4.13). In addition to this, Section 4.6.1 details how the ud shell was found to employ 91 nested eval(gzinflate(base64_decode())) constructs as a barrier to code inspection. All of these findings suggest a deliberate attempt to harden the sample against malware detectors and other forms of analysis.



(a) GUI of the mad1 shell (b) GUI of the mad2 shell

Figure 4.14: Interfaces of the mad1 and mad2 shell samples

4.6.5 Summary

The purpose of this case study was to demonstrate the functionality of the Matrix.py, Heatmap.py, and Dendrogram.py modules in a concise and meaningful way. Detailed analyses and interpretations of both the similarity matrices and their associated graphs were provided throughout to demonstrate how the different representations can be used to identify meaningful relationships between samples. The same approaches outlined in this case study can be readily applied to the more comprehensive tests detailed in Section 4.7.

During testing all of the available matrix combinations were employed to create the most comprehensive overview of sample similarity. The function name and function body measures focussed on PHP language features extracted from the sample source code, while the HTML similarity targeted the GUIs produced by each of the shells when run in a browser environment. The hashed chucks of each sample were also compared in an attempt to evaluate both the similarity of the HTML and the PHP code embedded within it. The results produced using these different measures provided some insight into the advantages and disadvantages of each approach.

As was expected, the results produced in Sections 4.6.1 and 4.6.2 during function name and function body matching were found to be very similar. Although the values recorded when comparing function bodies were lower on average than their function name counterparts, the

heatmaps produced by both measures of similarity demonstrated almost identical match distributions. The shape of the dendrogram representing relationships between samples based on their function body content also mimicked the shape generated when comparing function names, albeit with slightly weaker relationships.

The analysis of the HTML generated by each of the `c99` variants performed in Section 4.6.4 revealed two distinct clusters of similarity. These clusters were then evaluated through a visual inspection of the GUIs of the samples in each cluster. Section 4.6.3 drew conclusions about sample relationships based on an examination of fuzzy hashes generated from both the HTML present in each file and the PHP code embedded within it.

Regardless of the chosen method of analysis, higher levels of similarity were always observed when comparing samples that had been processed by the decoder. This trend was observed across similarity measures as a result of the increase in the amount of code that was made available for analysis. As a result of the decoding process, the average file size of the shells increased from 146.49 kB (for raw samples) to 151.27 kB (for their decoded counterparts). The presence of additional code allowed the batch modules to extract features that were previously unavailable and which could subsequently be compared to more accurately determine the similarity between two shell samples.

4.7 Comprehensive Tests

Although the smaller `c99` case study in the previous section is useful for demonstrating the similarity analysis process in a more concise and manageable manner, the goal of the system was to identify and interrogate areas of interest within larger datasets. Figure 4.15 shows the heatmap that was generated by the Heatmap.py module when performing function name matching among a random selection of 1000 decoded shells from the test collection. This number was chosen to preserve the resolution of the resulting image, as diagrams produced using the full collection of 2 129 samples were found to be illegible when fitted into an A4 page.

The diagram presented in Figure 4.15 provides an overview of the distribution of similarity within a collection of samples, but the sheer number of shells involved in the comparison meant that it lacked the resolution required for detailed analysis. For this reason, these large visualisation aids were most suited to the identification of general patterns and trends. In-depth analysis of the kind presented in the case study in Section 4.6 required that these comprehensive diagrams be split into more manageable and informative subsections. To demonstrate how this can be achieved, the following sections present two examples of how areas of interest can be identified, magnified, and analysed using each of the available visual representations of similarity.

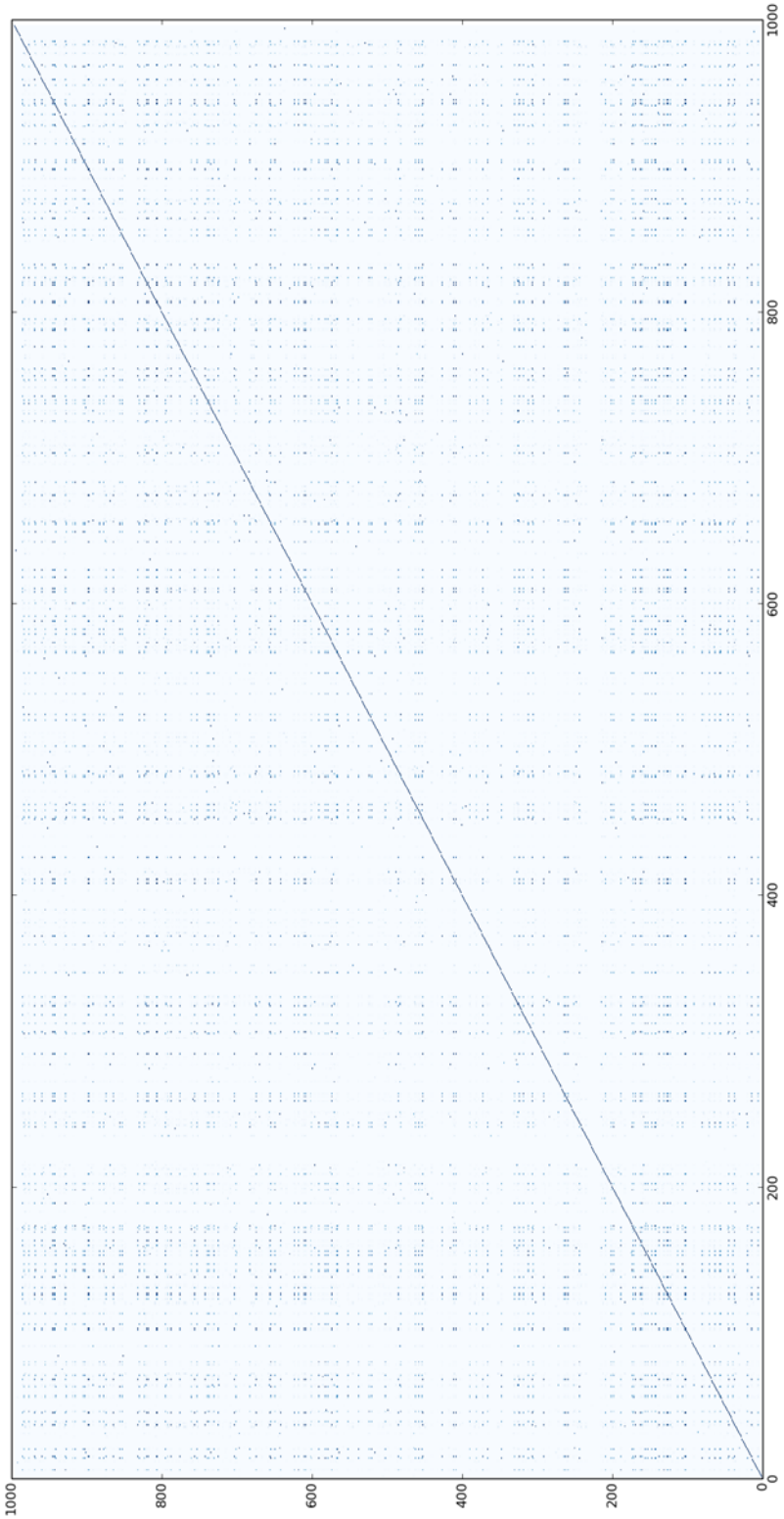


Figure 4.15: Heatmap based on the function names extracted from decoded shells

4.7.1 Heatmap Cluster Identification and Deobfuscation

For the first of the drill-down demonstrations, a random selection of 150 raw shells was used to create a large heatmap that could be used to identify areas of elevated similarity. The measure of similarity used in this instance was the percentage of matching function names, which drew on the function lists created by the Functions.py module. One similarity cluster was then identified and expanded by running the clustered samples through the decoder and then re-rendering the cluster to gauge any differences in observed similarity.

Figure 4.16 shows the heatmap that was obtained by running the 150 shells through the analysis process. Once this was completed, an area of interest (indicated by the selection pane towards the bottom left of Figure 4.16) was selected for the purpose of demonstration. An enlarged version of this area is displayed in Figure 4.17. In order to more accurately determine how similar this collection of samples was, all of the shells were run through the decoder, a new matrix was created, and a new heatmap was rendered, as demonstrated in Figure 4.18. A comparison of the heatmaps created before and after the deobfuscation process (shown in Figures 4.17 and 4.18, respectively) highlighted the improvement in similarity due to the increased availability of code for analysis.

The drill-down approach to similarity analysis had the added benefit of improving the efficiency of the system when targeting a single similarity cluster. Once an area of interest had been specified, only those samples that fell within it needed to be decoded, compared, and then re-rendered. Although the performance of the batch modules was found to scale linearly (as is described in Section 4.5), the number of comparisons performed by the Matrix.py module was proportional to the square of the number of samples in the collection. Re-rendering a smaller selection of samples instead of the entire collection contributed to a significant increase in performance and efficiency.

4.7.2 Dendrogram Relationship Identification

As discussed in Section 2.11.3, dendrogram structures are useful for visualising relationships between entities based on a given similarity metric. Figure 4.19 shows the dendrogram that was generated by the Dendrogram.py module when performing function name matching among the same selection of 1000 decoded shells used in Section 4.7.1.

As was the case with the heatmap in Section 4.7.1, Figure 4.19 lacks the resolution required for detailed analysis. Despite this, it can be used to identify relationship clusters that can subsequently be examined more closely. Three of the most distinct sample groupings are denoted by clusters X, Y, and Z. These three clusters are magnified in Figures 4.20, 4.21, and 4.22, respectively.

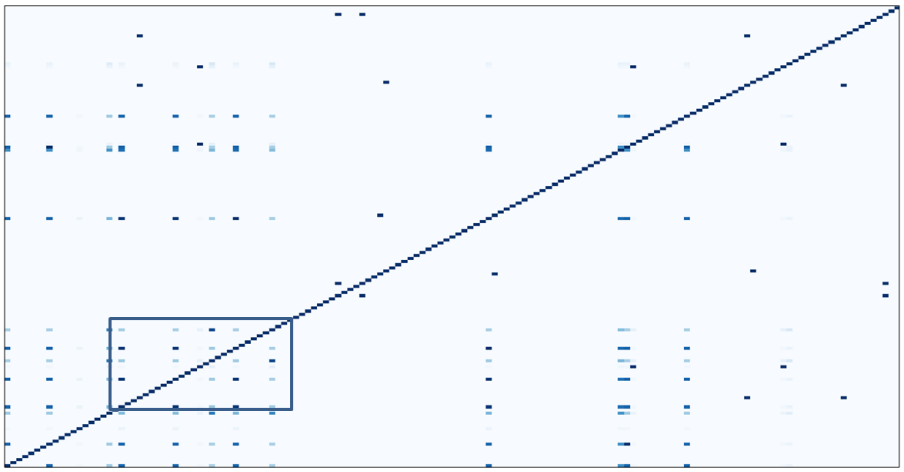


Figure 4.16: Similarity heatmap based on the function names extracted from a random selection of 150 raw shells

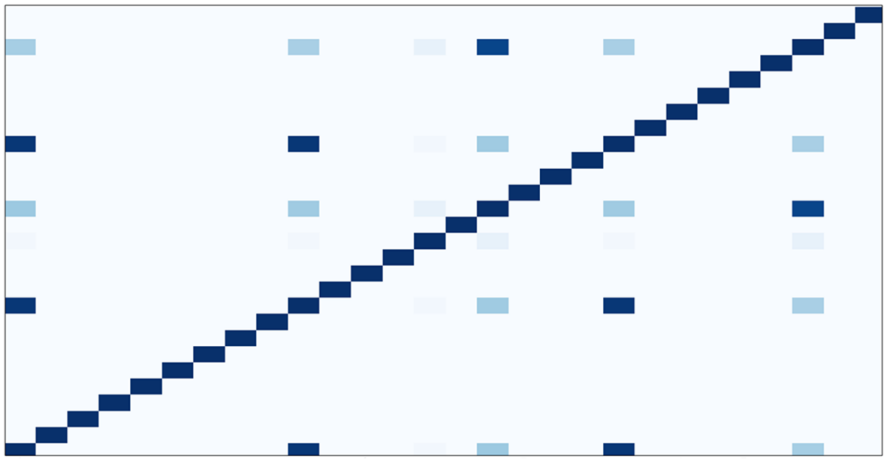


Figure 4.17: Focused similarity heatmap based on the cluster identified in Figure 4.16

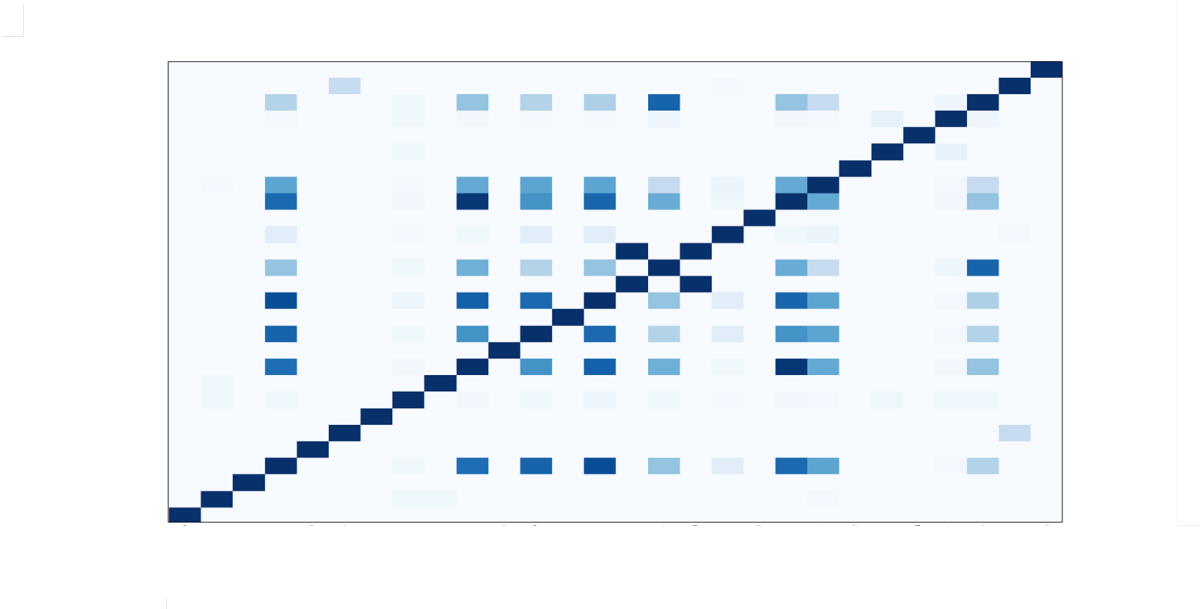


Figure 4.18: Similarity heatmap based on decoded version of the samples shown in Figure 4.17

Notwithstanding the sample naming inconsistencies discussed in Section 4.1, a brief examination of the x-axis labels of cluster X in Figure 4.20 suggested that the grouped samples were either derivatives of or vastly similar to the seminal `r57` shell in terms of user-defined function names. Although many of the samples were found to be identical according to this particular similarity measure (as is evidenced by the horizontal bars just above the x-axis), the presence of the tiered levels of similarity supported the hypothesis that new RATs are often created by modifying existing shells. In this case the detected modifications would have taken the form of either the addition or removal of user-defined functions. This is further supported by the depiction of cluster Y in Figure 4.21, which was found to contain tiered samples related to the `wso` shell. Although seemingly meaningful, the large grouping denoted by cluster A in Figure 4.19 was found to be a collection of otherwise unrelated samples that simply shared the same number of function name matches and were thus placed at the same level in the dendrogram.

The grouped samples of cluster Z in Figure 4.22 were all found to be related to the influential `c99` and `c100` shells based on the similarity of their user-defined function names. As two of the most widely-used RATs (Wardman *et al.*, 2009; Moore and Clayton, 2009), it is unsurprising that this collection was the largest of the clusters identified in Figure 4.19. Although several samples were found to share identical function names (as was observed among the `r57` derivatives in Figure 4.20), the fluctuating heights of the arches connecting shells once again suggested that many of them were created as variations of an existing RAT.

The horizontal bars above the x-axis denoting identical samples in Figures 4.20, 4.21, and 4.22 illustrate the limited scope of the function names measure of similarity when used in isolation. As explained in Section 4.1, each of the samples in the sample collection was tested to ensure uniqueness. If one were to use just the aforementioned figures as a reference, however, it would

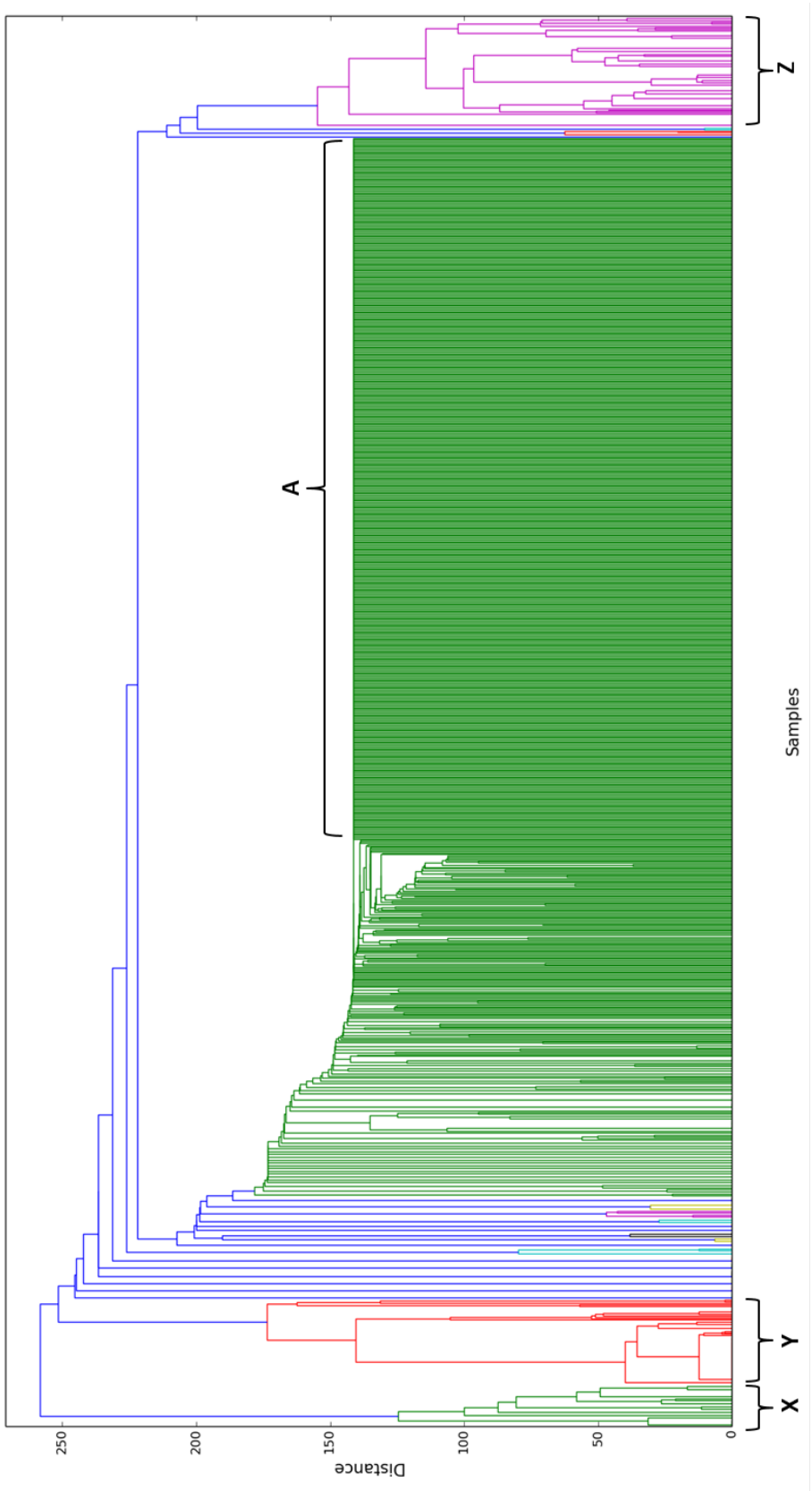


Figure 4.19: Dendrogram based on the function names extracted from 1000 decoded shells

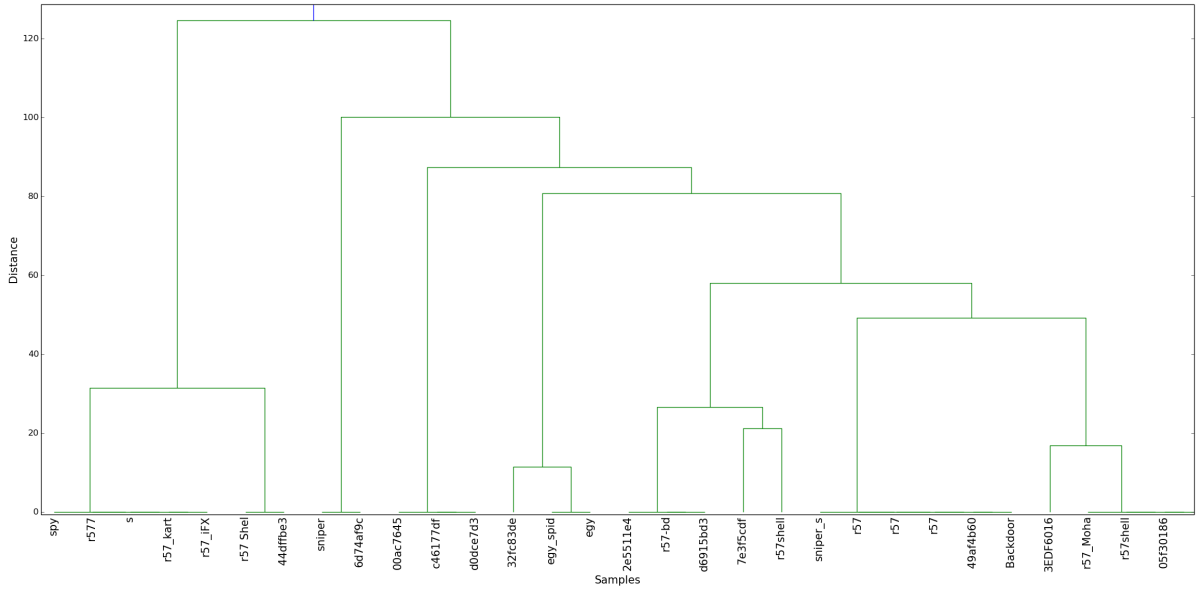


Figure 4.20: Focused dendrogram based on cluster X in Figure 4.19

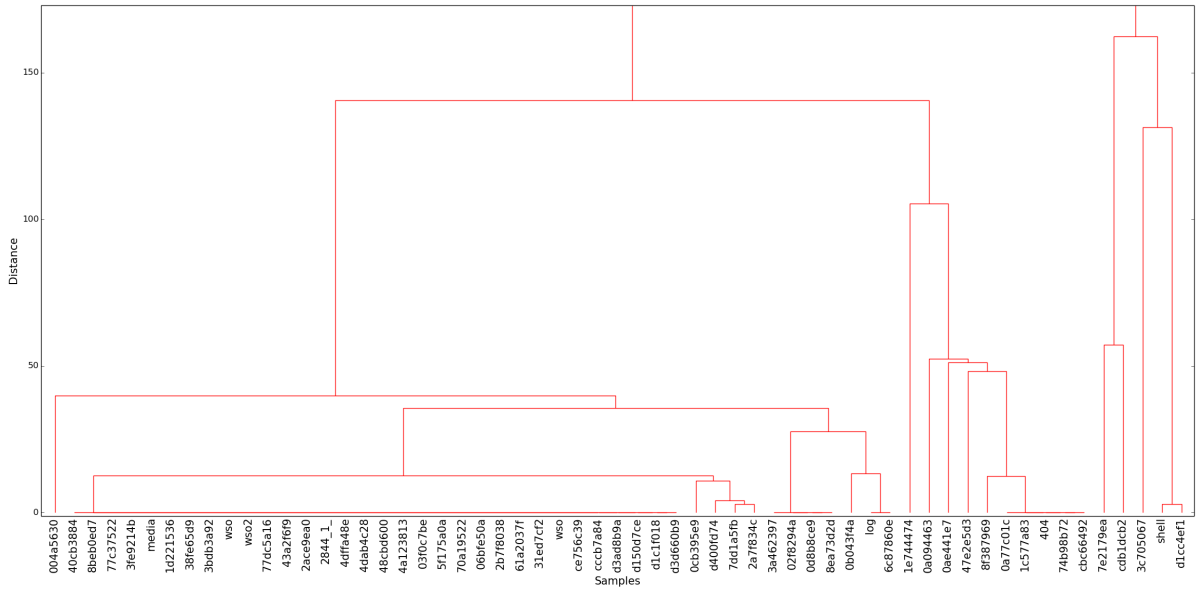


Figure 4.21: Focused dendrogram based on cluster Y in Figure 4.19

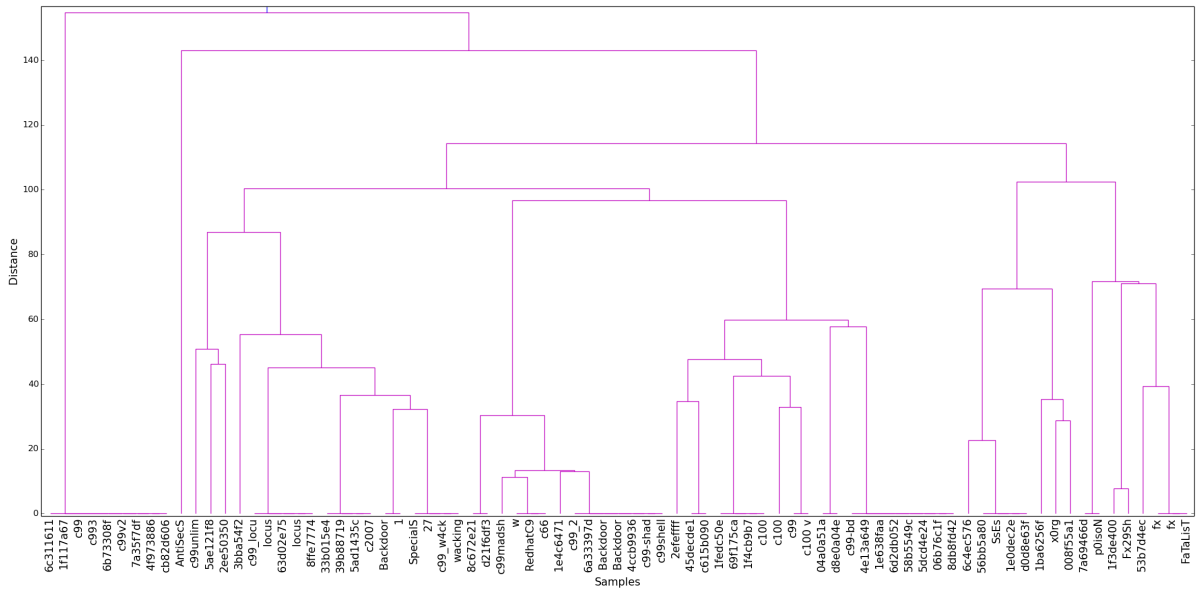


Figure 4.22: Focused dendrogram based on cluster Z in Figure 4.19

appear to contain several duplicates. In the absence of a system that combines the outputs produced using different measures of similarity (and which is discussed as a possible extension to this research in Section 5.3), it is therefore important that function name matching be considered in conjunction with the other measures listed in Section 3.5.

4.7.3 Summary

The purpose of the comprehensive tests presented in this section was to demonstrate how similarity analysis can be performed on larger datasets. To accomplish this, the Heatmap.py and Dendrogram.py modules were used to create diagrams representing the function name matches between 1000 randomly selected RATs. These diagrams were later interpreted and analysed to determine similarity trends between samples in the collection.

Although both the heatmap and dendrogram proved useful for identifying high-level features (such as areas of high similarity and clusters of related shells), the sheer size of both diagrams obscured the detail required for more accurate analysis. In both cases, specific sections of the larger diagrams were therefore extracted, magnified, and – in the case of the heatmap – re-rendered to increase the resolution to a functional level. Once this had been achieved, analysis similar to that presented in the case study in Section 4.6 could be successfully carried out.

Table 4.6: Batch modules similarity statistics for raw samples

	Total	Average	Variance	Std Deviation
Function names	10243768	2.26	1.57	1.25
Function bodies	4306008	0.95	1.64	1.28
Hashed chunks	5303189	1.17	0.24	0.49
HTML	1949035	0.43	0.78	0.88

Table 4.7: Batch modules similarity statistics for decoded samples

	Total	Average	Variance	Std Deviation
Function names	10243768.66	2.69	1.53	1.28
Function bodies	4306008.95	1.09	1.66	1.29
Hashed chunks	5303189.97	1.53	0.24	0.49
HTML	90652.82	0.45	0.79	0.89

4.8 Evaluation of Similarity Measures

Each of the similarity measures represented by the individual modules described in Section 3.5 target specific features of PHP-based RATs. Both the Function.py and FunctionBodies.py modules extract information about function constructs, the HtmlDump.py module targets embedded HTML, and the HashChunks.py module processes a file in its entirety. Although the accuracy of each of these measures is difficult to determine without a reference set of similarity values, it is nonetheless useful to compare the results produced by each of these measures when run against the test collection outlined in Section 4.1.

Tables 4.6 and 4.7 summarise the results obtained by each of the four measures of similarity when run against raw and decoded samples, respectively. As was expected, the average similarity values detected by all of these measures were significantly lower than the levels that were observed when analysing seven of the `c99` derivatives in Section 4.6, as those samples were known to be related prior to analysis and are not representative of the entire dataset. Despite this, many of the trends discovered during the `c99` case study were found to persist when analysing the results obtained from the full sample collection.

As was the case when comparing `c99` derivatives in Section 4.6, the process of deobfuscating and normalising shell samples prior to analysis had the effect of increasing the observed levels of similarity for all similarity measures. A comparison of the average similarities recorded by each of the different measures for raw and decoded sample sets demonstrated a minimum increase of 4.65% (when comparing HTML dumps), and a maximum increase of 30.77% (when comparing hashed file chunks). The similarity levels across the different measures increased by an average of 17.30%. These increases clearly demonstrate the benefits of both the deobfusaction and format normalisation performed by the decoder component described in Section 3.3.

The seemingly illogical disparity between the function name and function body measures of similarity (which was also observed during the case study described in Section 4.6) can be attributed to a combination of three possible scenarios:

1. The concise and generic nature of function names means that comparisons between them are more likely to return false positives (i.e., functions that share the same name but have vastly different purposes and implementations). This is far less likely to occur when comparing function bodies, which contain far more content and are thus less likely to be mistakenly correlated.

2. Although malware authors are known to reuse a substantial amount of code (including functions) when creating newer versions (Walenstein *et al.*, 2007; Nair *et al.*, 2010; Walenstein and Lakhotia, 2007), the implementations of these functions could be modified during the update process, resulting in lower implementation similarity.

3. The block size limitation of the Ssdeep fuzzy hashing algorithm discussed in Sections 2.10.4 and 4.6.2 causes function bodies with less than 200 characters to return an artificially low similarity score. As a result of this, some identical function implementations may have been recorded as being only partially similar, thereby reducing the overall function body similarity scores.

4.9 Chapter Summary

This chapter presented the results that were obtained during the testing of the various system components. It began with an evaluation of the decoder, which was found to be capable of removing obfuscation constructs ranging from rudimentary `eval()` and `preg_replace()` functions to more complex structures with multi-layered obfuscation and auxiliary string manipulation functions. The individual modules were used to process a derivative of the `r57` shell to test their feature-extraction capabilities. A similarity case study of the `c99` family of shells was presented to demonstrate how the diagrams produced by the matrix and visualisation modules can be interpreted to identify meaningful inter-sample relationships. Finally, two large-scale tests were performed to demonstrate how areas of interest can be isolated and magnified to identify similarity clusters within a large dataset.

5

Conclusion

The abundance of PHP-based RATs found in the wild has led malware researchers to develop systems capable of tracking and analysing these shells (Bailey *et al.*, 2007; Baxter *et al.*, 1998; Chouchane *et al.*, 2007). In the past, shell relationships were ably identified using signature matching, a process that is currently unable to cope with the sheer volume and variety of web-based malware in circulation. Although a large percentage of newly-created webshell software incorporates portions of code derived from seminal shells such as `c99` and `r57`, they are able to disguise this by making extensive use of obfuscation techniques intended to frustrate any attempts to dissect or reverse engineer the code.

In response to the problem outlined above, this thesis began in Chapter 2 by contextualising the research and providing an overview of background information in the fields of code obfuscation and similarity analysis. This included a discussion of relevant features of the PHP language, as well as a description of the basic structure and capabilities of a typical web shell. An overview of the various methods of obfuscating code was also presented, with particular emphasis on idiomatic code hiding constructs that are unique to RATs written in PHP. An evaluation of several common approaches to software similarity analysis followed, including the concept of fuzzy or approximate hashing. Next three techniques for creating intuitive visual representations of the results of similarity analysis and hierarchical clustering were described. The chapter concluded with a critical overview of work already undertaken in the fields of code deobfuscation and similarity analysis.

Chapter 3 presented the design and implementation of the system responsible for the detection of derivative malware samples written in PHP. A high-level overview of the system's structure was presented, including an description of how test samples are passed from one component to another. The two download scripts responsible for retrieving and storing these files were then introduced, as was the decoder component, which is used to deobfuscate and normalise sample inputs prior to analysis. Viper, the malware analysis framework that was used as the basis for the similarity analysis system, was described thereafter. This was followed by an outline of the individual and batch modules that were used to extract pertinent features for subsequent comparison by the matrix module. The chapter concluded with a description of the two modules used to create visualisations of the results produced by the Matrix.py module for the purpose of identifying meaningful inter-sample relationships.

The results of the extensive system testing that was undertaken to determine the efficacy of each component were presented in Chapter 4. First, the decoder component was subjected to a range of tests designed to evaluate its ability to remove a variety of obfuscation constructs. This was followed by an analysis of the results produced by the individual modules when extracting comparable features from a derivative of the seminal `r57` web shell. The Matrix.py, Heatmap.py, and Dendrogram.py modules were critically evaluated through the use of a case study involving the `c99` family of shells. The chapter concluded by demonstrating the system's ability to simplify and re-render large complex visualisations to identify clusters of related samples in greater detail.

As outlined in Chapter 1, the five main goals of this research were as follows:

1. The creation of a decoder component capable of normalising and deobfuscating test samples prior to similarity analysis. The purpose of this component would be to reverse commonly-used obfuscation idioms, thereby exposing more code for analysis.

2. The construction of four separate preprocessing modules designed to extract relevant features for comparison.

3. The implementation of a modular system designed to compare the features extracted by the preprocessing modules and create representative similarity matrices.

4. The creation of two visualisation modules capable of creating graphic representations of the results obtained during similarity analysis for ease of interpretation. The purpose of these modules is to facilitate the identification of meaningful relationships among samples by analysts.

5. An evaluation of the effects of the deobfuscation process on the results produced during similarity analysis.

Chapter 4 demonstrated the extent to which each of these goals was met:

1. Testing performed in Section 4.2 demonstrated the decoder component's ability to remove a wide variety of idiomatic obfuscation constructs, ranging from simple `eval()` and `preg_replace()` functions to multi-layered obfuscation constructs containing auxiliary string manipulation functions. This deobfuscation process was successfully applied to the full collection of 2 129 test samples in preparation for similarity analysis.

2. The results of the individual module tests presented in Section 4.4 showed that each module was capable of correctly extracting and recording their respective sample features. Function names and bodies were ably located by the Function.py and FunctionBodies.py modules, while the HashChunks.py and HtmlDump.py modules correctly extracted hashed file chunks and embedded sections of HTML, respectively. All results were confirmed via manual observation to ensure their accuracy.

3. The comprehensive case study outlined in Section 4.6 was used to test the capabilities of the remaining Matrix.py, Heatmap.py, and Dendrogram.py modules when run against a restricted sample set, and to demonstrate how these results can be interpreted to identify meaningful relationships between samples. The Matrix.py module was successfully used to create representative similarity matrices for each of the four similarity measures. Its capabilities were further tested during the large-scale testing undertaken in Section 4.7, where it was able to create more complex matrices containing one million individual match values.

4. The two visualisation modules proved to be useful for providing a comprehensive overview of similarity, even in large datasets. Sections 4.6 and 4.7 both demonstrated how these modules can be used to create intuitive visualisations that enable the viewer to easily identify similarity trends for a given set of malware samples. Regions of high similarity were most ably highlighted by the heatmap diagrams, while sample relationships (and their respective magnitudes) were most easily identified through a consultation of the dendrograms generated in each case.

5. It was found that the deobfuscation performed by the decoder component prior to analysis dramatically increased the observed levels of similarity between test samples. This was first made apparent during the case study undertaken in Section 4.6. In some cases, heavily obfuscated shells that initially displayed little similarity to other samples were found to be almost identical when analysed in their decoded forms. The primary reason for this improvement in matching accuracy was found to be the sharp increase in the amount of code available for use by the remainder of the similarity analysis system. Further proof of the efficacy of the decoder component was presented in Section 4.8, where it was shown to improve the observed levels of similarity of all measures by an average of 17.30%.

In summary, this research has achieved the goals outlined in Section 1.2. A malware analysis system that is capable of using a variety of similarity measures to determine derivative relationships between RATs written in PHP was successfully developed and tested. In addition to this, it was determined that the novel pairing of such a system with a decoder component able to detect and reverse idiomatic obfuscation constructs vastly increased the accuracy of the results by exposing additional code for analysis.

5.1 Secondary Outcomes

In addition to fulfilling of the primary research goals, the process of designing and testing a system capable of identifying derivative shell relationships produced a number of secondary outcomes and observations:

- The use of multiple measures of similarity allowed for a more thorough overview of inter-sample relationships. Both the `compare_funcs()` and `compare_bodies()` functions were used to identify oft-copied PHP language features, while the `compare_html()` function was able to highlight the resemblance between HTML-based shell GUIs. The `compare_chunks()` function attempted to evaluate the overall similarity among sample pairs by splitting both the PHP source code and the embedded HTML into chunks and comparing their computed fuzzy hashes. By consulting the diagrams produced using each of these four measures, a more complete picture of similarity could be formed.

- Techniques for isolating and re-rendering specific clusters of similarity from within a larger diagram were introduced. The sharp increases in resolution observed when dealing with these smaller clusters enabled in-depth analysis of the kind described while comparing the `c99` family of shells. This drill-down approach to similarity analysis had the added benefit of improving the efficiency of the system as a whole, as only those samples that fell within an area of interest needed to be decoded, compared, and then re-rendered. Although the performance of the batch modules scaled linearly, the number of comparisons performed by the Matrix.py module was found to be proportional to the square of the number of samples in the collection. Re-rendering a smaller selection of samples instead of the entire collection thus contributed to a significant increase in performance and efficiency in each case.

- Testing of the similarity analysis system provided several insights into the modifications commonly made by authors when creating new malware variants. Statistics recorded by the decoder provided evidence of idiomatic code-hiding constructs, while output produced during HTML comparisons of heavily obfuscated samples such as the `c99_ud.txt` shell proclaimed them to be certified as “antivirus undetected”, another clear sign of deliberate obfuscation. An analysis of shell relationships post-deobfuscation also confirmed the

existence of several variants of the influential shells such as `c99` and `r57`, among others. The GUIs of these variants were often found to be very similar, with most of the changes occurring in the underlying implementation. Slight differences between the GUIs of related shells were found to be largely cosmetic and ego-driven in nature – many authors simply made changes to colour schemes, logos, and version names, or added signatures crediting the individuals/groups responsible for modifying the shell.

Several of the tools and techniques developed during the course of this research are unique and constitute an addition to the existing body of knowledge relating to malware analysis. To the best of the author's knowledge, no PHP-specific malware source code comparison tools have been created and successfully tested. The combination of deobfuscation, similarity analysis, and subsequent visualisation provides a useful and thorough overview of the similarity of these types of files, and the use of multiple measures of similarity allows for a less one-sided approach to malware analysis. The combination of visualisation techniques allows similarity to be interpreted instead of merely acknowledged, and the insights into malware derivation that resulted from this research open up new areas for further research in the fields of malware evolution and adaptation.

5.2 Limitations

Throughout the course of this research, especially during the testing process, several limitations of the current system implementation were identified. Some of these limitations exist outside the scope that was originally defined for this research and can therefore be ignored, while others are recommended as potential opportunities for future research in Section 5.3.

The most common misconception that was expressed by reviewers of the published portions of this work was that it was intended as an antivirus solution. As detailed in Chapter 1, the goal of this research was not to identify new shell variants, but rather to pinpoint derivative relationships within a collection of known malware. For this reason, no testing was done to ascertain how adept the similarity analysis system was at determining whether a given sample was malicious. Although a high level of similarity to a known malware sample might be indicative of malicious behaviour, further analysis would have to be done to make an informed decision. For this reason, the system could be used in conjunction with established malware detectors to flag samples as being suspicious and worthy of further investigation, but not as a standalone antivirus solution.

One of the major limitations encountered during the testing of the system was a lack of reference similarity values with which to compare the obtained results. This lack of reference data arose as a result of the uniqueness of both the measures of similarity and the collection of samples used for the research. Despite this, a combination of rigorous component testing and the manual verification of observed relationships and patterns contributed to the certainty of the outcomes of the research.

The block size and minimum length limitations associated with the Ssdeep hashing algorithm (described in Section 2.10) affected the results produced by the system when using either the function body or HTML output measures of similarity. Function bodies with less than the required 200 characters were found to return artificially low similarity scores as a result of the length limitation, while HTML dumps that differed substantially in length returned a similarity score of zero as a result of the block size limitation. As a result of these observed discrepancies, Section 5.3 recommends the implementation of an alternative fuzzy hashing algorithm as one of the potential improvements to the current system.

5.3 Future Work

During the course of this research, several opportunities for future work were identified:

- Although the four modules described in Section 3.5 proved useful as measures of similarity, they represent only a few approaches to the detection of code reuse in web shells. In future, a thorough evaluation of alternate classification methods could be carried out to determine which approach (or combination of approaches) is most accurate. The following methods could be considered:
  - Control graph matching
  - Dynamic sandbox analysis
  - Line-by-line analysis
  - N-gram analysis
  - Normalised compression distance
- According to Flynn's taxonomy (Chalmers *et al.*, 2002), the decoder, the batch modules, and the Matrix.py module can all be classified as single instruction sets with multiple data streams (in this case each of the sample inputs)(Chalmers *et al.*, 2002). No synchronisation is required between the processing of each sample, and as such the performance of these components would benefit greatly from the use parallel programming techniques. An investigation into the use of Graphical Processing Units (GPUs) for these particular tasks could also be undertaken in an attempt to improve the performance of the similarity analysis system as a whole.
- Although useful for creating comparable fuzzy hashes, the input length and block size limitations of the Ssdeep approximate hashing algorithm prevent it from producing accurate results in all situations. For this reason, other approximate hashing algorithms such as sdhash (Roussev, 2015) could be investigated as possible replacements. Studies undertaken by Vassil Roussev have indicated that this algorithm performs better in terms of both precision and scalability (Roussev, 2012, 2011).

Glossary

API Application Program Interface

ASP Active Server Pages

AST Abstract Syntax Tree

B Bytes

CMS Content Management System

CTPH Context-Triggered Piecewise Hashing

DDOS Distributed Denial of Service

DNA Deoxyribonucleic Acid

FTP File Transfer Protocol

GUI Graphical User Interface

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

IP Internet Protocol

kB Kilobyte

JSON JavaScript Object Notation

LAMP Linux, Apache, MySQL and PHP/Perl/Python

MB Megabytes

MD5 Message Digest 5

PC Personal Computer

PHP Hypertext Preprocessor

RAT Remote Access Trojan

SHA Secure Hash Algorithm

SQL Structured Query Language

URL Uniform Resource Locator

XXS Cross-site Scripting

References

Abou-Assaleh, T., Cercone, N., Kešelj, V., and Sweidan, R. *N-gram-based detection of new malicious code.* In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 2, pages 41–42. IEEE, 2004.

Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

Argerich, L. *Professional PHP4.* Professional Series. Wrox Press, 2002. ISBN 9781861006912.

Atkinson, L. and Suraski, Z. *Core PHP Programming.* Core series. Prentice Hall Computer, 2004. ISBN 9780130463463.

AV Test. *Total malware.* Statistical reference website, 2015. Accessed: 12 November 2015. URL `https://www.av-test.org/en/statistics/malware/`

Baier, H. and Breitinger, F. *Security aspects of piecewise hashing in computer forensics.* In *IT Security Incident Management and IT Forensics (IMF), 2011 Sixth International Conference on*, pages 21–36. IEEE, 2011. doi:10.1109/IMF.2011.16.

Bailey, M., Oberheide, J., Andersen, J., Mao, Z. M., Jahanian, F., and Nazario, J. *Automated classification and analysis of internet malware.* In *Recent Advances in Intrusion Detection*, pages 178–197. Springer, 2007.

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. *On the (im)possibility of obfuscating programs.* In *Advances in Cryptology-CRYPTO 2001*, pages 1–18. Springer, 2001. doi:10.1145/2160158.2160159.

Baxter, I. D., Yahin, A., Moura, L., Anna, M. S., and Bier, L. *Clone detection using abstract syntax trees.* In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.

Bayer, U., Comparetti, P. M., Hlauschek, C., Kruegel, C., and Kirda, E. *Scalable, behavior-based malware clustering.* In *Network and Distributed System Security*, volume 9, pages 8–11. Citeseer, 2009.

Berdajs, J. and Bosnic, Z. *Extending applications using an advanced approach to DLL injection and API hooking. Software: Practice and Experience*, 40(7):567–584, 2010. ISSN 1097-024X. doi:10.1002/spe.973.

Binkley, D. *Source Code Analysis: A Road Map.* In *2007 Future of Software Engineering*, FOSE '07, pages 104–119. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2829-5. doi:10.1109/FOSE.2007.27.
URL `http://dx.doi.org/10.1109/FOSE.2007.27`

Borello, J.-M. and Mé, L. *Code obfuscation techniques for metamorphic viruses. Journal in Computer Virology*, 4(3):211–220, 2008.

Bressert, E. *SciPy and NumPy: An Overview for Developers.* " O'Reilly Media, Inc.", 2012. ISBN 1449305466.

Brodlie, K., Osorio, R. A., and Lopes, A. *A review of uncertainty in data visualization.* In *Expanding the Frontiers of Visual Analytics and Visualization*, pages 81–109. Springer, 2012.

Bughin, J., Chui, M., and Johnson, B. *The next step in open innovation. The McKinsey Quarterly*, 4(6):1–8, 2008.

Buja, A., Cook, D., and Swayne, D. F. *Interactive high-dimensional data visualization. Journal of Computational and Graphical Statistics*, 5(1):78–99, 1996.

Campanella, J. J., Bitincka, L., and Smalley, J. *MatGAT: an application that generates similarity/identity matrices using protein or DNA sequences. BMC bioinformatics*, 4(1):29, 2003.

Cecchet, E., Chanda, A., Elnikety, S., Marguerite, J., and Zwaenepoel, W. *Performance Comparison of Middleware Architectures for Generating Dynamic Web Content.* In **Endler, M. and Schmidt, D.**, editors, *Middleware 2003*, volume 2672 of *Lecture Notes in Computer Science*, pages 242–261. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40317-3. doi:10.1007/3-540-44892-6_13.

Chalmers, A., Reinhard, E., and Davis, T. *Practical parallel rendering.* CRC Press, 2002. ISBN 1568811799.

Chen, L. and Wang, G. *An Efficient Piecewise Hashing Method for Computer Forensics.* In *Knowledge Discovery and Data Mining, 2008. WKDD 2008. First International Workshop on*, pages 635–638. Jan 2008. doi:10.1109/WKDD.2008.80.

Choi, S.-S., Cha, S.-H., and Tappert, C. C. *A survey of binary similarity and distance measures. Journal of Systemics, Cybernetics and Informatics*, 8(1):43–48, 2010.

Cholakov, N. *On some drawbacks of the PHP platform.* In *Proceedings of the 9th International Conference on Computer Systems and Technologies and Workshop for PhD Students in*

Computing, CompSysTech '08, pages 12:II.7–12:2. ACM, New York, NY, USA, 2008. ISBN 978-954-9641-52-3. doi:10.1145/1500879.1500894.

Chouchane, M. R. and Lakhotia, A. *Using engine signature to detect metamorphic malware.* In *Proceedings of the 4th ACM workshop on Recurring malcode*, pages 73–78. ACM, 2006. doi:10.1145/1179542.1179558.

Chouchane, M. R., Walenstein, A., and Lakhotia, A. *Statistical signatures for fast filtering of instruction-substituting metamorphic malware.* In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 31–37. ACM, 2007. doi:10.1145/1314389.1314397.

Christodorescu, M. and Jha, S. *Testing malware detectors. ACM SIGSOFT Software Engineering Notes*, 29(4):34–44, 2004.

Christodorescu, M., Jha, S., Kinder, J., Katzenbeisser, S., and Veith, H. *Software transformations to improve malware detection. Journal in Computer Virology*, 3(4):253–265, 2007. ISSN 1772-9890. doi:10.1007/s11416-007-0059-8.

Christodorescu, M., Kinder, J., Jha, S., Katzenbeisser, S., and Veith, H. *Malware normalization.* Technical report, University of Wisconsin, 2005.

Chuvakin, A. *An overview of unix rootkits. iALERT White Paper, iDefense Labs*, 2003. Accessed: 2 June 2015.
URL `http://www.megasecurity.org/papers/Rootkits.pdf`

Coelho, F. *PHP-related vulnerabilities on the National Vulnerability Database.* 2013. Accessed on 25 May 2015.
URL `http://www.coelho.net/php-cve.html`

Collberg, C., Thomborson, C., and Low, D. *A taxonomy of obfuscating transformations.* Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

Decloedt, H. E. and van Heerden, R. *Rootkits, trojans, backdoors and new developments. Proceedings of the Workshop on ICT Uses in Warfare and the Safeguarding of Peace*, pages 4–11, 2010.

Denning, P. J. *The locality principle. Communications of the ACM*, 48(7):19–24, 2005.

Doyle, M. *Beginning PHP 5.3.* Wiley, 2011. ISBN 9781118057346.

Edem, E. I., Benzaid, C., Al-Nemrat, A., and Watters, P. *Analysis of malware behaviour: Using data mining clustering techniques to support forensics investigation.* In *Cybercrime and Trustworthy Computing Conference (CTC), 2014 Fifth*, pages 54–63. IEEE, 2014.

Ertaul, L. and Venkatesh, S. *Jhide - a tool kit for code obfuscation.* In *8th IASTED International Conference on Software Engineering and Applications (SEA 2004)*, pages 133–138. 2004.

Everitt, B. S. and Skrondal, A. *The Cambridge Dictionary of Statistics.* Cambridge University Press, 2002. ISBN 0521766990.

Fayyad, U. M., Wierse, A., and Grinstein, G. G. *Information visualization in data mining and knowledge discovery.* Morgan Kaufmann, 2002. ISBN 1558606890.

Friendly, M. and Denis, D. J. *Milestones in the history of thematic cartography, statistical graphics, and data visualization.* 2001. Accessed: 12 February 2014.
URL `http://www.datavis.ca/milestones`

Guarnieri, C. *Creating New Modules.* Framework reference website, 2015a. Accessed: 7 May 2015.
URL `http://viper-framework.readthedocs.org/en/latest/customize/index.html#create-new-modules`

Guarnieri, C. *Viper Commands.* Framework reference website, 2015b. Accessed: 16 April 2015.
URL `http://viper-framework.readthedocs.org/en/latest/usage/commands.html`

Guarnieri, C. *Viper Official Documentation.* Framework reference website, 2015c. Accessed: 16 April 2015.
URL `http://viper-framework.readthedocs.org/en/latest/index.html`

Guarnieri, C. *Viper Projects.* Framework reference website, 2015d. Accessed: 16 April 2015.
URL `http://viper-framework.readthedocs.org/en/latest/usage/concepts.html#projects`

Guarnieri, C. *Viper Sessions.* Framework reference website, 2015e. Accessed: 16 April 2015.
URL `http://viper-framework.readthedocs.org/en/latest/usage/concepts.html#sessions`

Haagman, D. and Ghavalas, B. *Trojan Defence: A Forensic View. Digital Investigation*, 2(1):23–30, 2005.

Hartigan, J. A. *Representation of similarity matrices by trees. Journal of the American Statistical Association*, 62(320):1140–1158, 1967. doi:10.1080/01621459.1967.10500922.

Hunter, J. and Dale, D. *The matplotlib users guide.* 2007. Accessed: 30 March 2015.
URL `http://matplotlib.org/users/index.html`

Hunter, J. D. *Matplotlib: A 2d graphics environment. Computing in Science and Engineering*, 9(3):90–95, 2007.

Hutchins, E. M., Cloppert, M. J., and Amin, R. M. *Intelligence-driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains. Leading Issues in Information Warfare & Security Research*, 1:80, 2011.

Iam-on, N., Boongoen, T., and Garrett, S. *Refining pairwise similarity matrix for cluster ensemble problem with cluster relations.* In **Jean-Fran, J.-F., Berthold, M., and Horvth, T.**, editors, *Discovery Science*, volume 5255 of *Lecture Notes in Computer Science*, pages 222–233. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-88410-1. doi: 10.1007/978-3-540-88411-8_22.

Ide, A. *PHP Just Grows and Grows.* Technical survey website, 2015. Accessed: 18 April 2015. URL http://news.netcraft.com/archives/2013/01/31/php-just-grows-grows.html

Jang, J., Brumley, D., and Venkataraman, S. *Bitshred: feature hashing malware for scalable triage and semantic analysis.* In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 309–320. ACM, 2011. doi:10.1145/2046707.2046742.

Kaspersky, E. *Number of the Month: 70K per day.* October 2011. Accessed on 1 March 2013. URL http://eugene.kaspersky.com/2011/10/28/number-of-the-month-70k-per-day/

Katz, O. *Detecting remote file inclusion attacks. White Paper. Breach Security Inc., May*, 2009. Accessed: 14 May 2015.
URL https://www.owasp.org/images/6/67/OWASP_Israel_-_March_2009_-_Or_Katz_-_RFI_detection.pdf

Kazanciyan, R. *Old Web Shells, New Tricks.* December 2012. Accessed: 14 June 2013.
URL https://www.owasp.org/images/c/c3/ASDC12-Old_Webshells_New_Tricks_How_Persistent_Threats_haverevived_an_old_idea_and_how_you_can_detect_them.pdf

Kienzle, D. M. and Elder, M. C. *Recent worms: A survey and trends.* In *Proceedings of the 2003 ACM Workshop on Rapid Malcode*, WORM '03, pages 1–10. ACM, New York, NY, USA, 2003. ISBN 1-58113-785-0. doi:10.1145/948187.948189.

Kornblum, J. *Identifying almost identical files using context triggered piecewise hashing. Digital Investigation*, 3:91–97, 2006.

Kornblum, J. *Context Triggered Piecewise Hashes.* July 2013. Accessed on 26 October 2013. URL http://ssdeep.sourceforge.net/

Landesman, M. *Malware Revolution: A Change in Target.* March 2007. Accessed: 16 June 2014.
URL `http://technet.microsoft.com/en-us/library/cc512596.aspx`

Li, J., Xu, M., Zheng, N., and Xu, J. *Malware obfuscation detection via maximal patterns.* In *Intelligent Information Technology Application, 2009. IITA 2009. Third International Symposium on*, volume 2, pages 324–328. IEEE, 2009.

Linn, C. and Debray, S. *Obfuscation of Executable Code to Improve Resistance to Static Disassembly.* In *ACM Conference on Computer and Communications Security*, pages 290–299. ACM Press, 2003.

Lu, G. and Debray, S. *Automatic Simplification of Obfuscated JavaScript Code: A Semantics-Based Approach.* In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 31–40. June 2012. doi:10.1109/SERE.2012.13.

Madou, M., Van Put, L., and De Bosschere, K. *Loco: An interactive code (de)obfuscation tool.* In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 140–144. ACM, New York, NY, USA, 2006. ISBN 1-59593-196-1. doi:10.1145/1111542.1111566.

McKinney, W. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* O'Reilly Media, Inc., 2012. ISBN 1449319793.

McLaughlin, B. *PHP & MySQL.* Missing Manual. O'Reilly Media, Incorporated, 2012. ISBN 9781449325572.

Miller, R. *PHP Apps A Growing Target for Hackers.* January 2006. Accessed on 25 May 2013.
URL `http://news.netcraft.com/archives/2006/01/31/php_apps_a_growing_target_for_hackers.html`

Moore, T. and Clayton, R. *Evil searching: Compromise and Recompromise of Internet Hosts for Phishing.* In *Financial Cryptography and Data Security*, pages 256–272. Springer, 2009. ISBN 978-3-642-03548-7.

Moore, T. W. and Clayton, R. *The impact of public information on phishing attack and defense. Communications and Strategies*, 81:45–68, 2011.

Nair, V. P., Jain, H., Golecha, Y. K., Gaur, M. S., and Laxmi, V. *Medusa: Metamorphic malware dynamic analysis usingsignature from api.* In *Proceedings of the 3rd International Conference on Security of Information and Networks*, SIN '10, pages 263–269. ACM, New York, NY, USA, 2010. doi:10.1145/1854099.1854152.

Oliphant, T. E. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006. ISBN 151730007X.

Open Source Matters. *What is Joomla?* January 2013. Accessed on 25 May 2013.
URL `http://www.joomla.org/about-joomla.html`

Pfleeger, C. P. and Pfleeger, S. L. *Security in Computing*. Prentice Hall Professional Technical Reference, 2002. ISBN 0132390779.

PHP Group. *Output Control*. Library reference website. Accessed on 16 August 2015.
URL `http://php.net/manual/en/book.outcontrol.php`

PHP Group. *Basic Syntax*. Library reference website, 2015a. Accessed on 22 May 2015.
URL `http://php.net/manual/en/language.basic-syntax.php`

PHP Group. *Eval*. Library reference website, 2015b. Accessed: 2 March 2015.
URL `http://php.net/manual/en/function.eval.php`

PHP Group. *Function Reference*. Library reference website, 2015c. Accessed on 22 May 2015.
URL `http://www.php.net/manual/en/funcref.php`

PHP Group. *Installation and Configuration*. Library reference website, 2015d. Accessed on 22 May 2015.
URL `http://www.php.net/manual/en/install.php`

PHP Group. *PEAR - PHP Extension and Application Repository*. Library reference website, 2015e. Accessed on 4 July 2015.
URL `http://pear.php.net/`

PHP Group. *PECL*. Library reference website, 2015f. Accessed on 22 May 2015.
URL `http://pecl.php.net/`

PHP Group. *Preg Replace*. Library reference website, 2015g. Accessed: 2 March 2015.
URL `http://php.net/manual/en/function.preg-replace.php`

PHP Group. *Tokenizer*. Library reference website, 2015h. Accessed: 2 March 2015.
URL `http://php.net/manual/en/intro.tokenizer.php`

PHP Group. *Usage Stats for January 2013*. Library reference website, 2015i. Accessed on 8 October 2015.
URL `http://php.net/usage.php`

PHP Group. *What can PHP do?* Library reference website, 2015j. Accessed on 15 February 2015.
URL `http://www.php.net/manual/en/intro-whatcando.php`

PHP Group. *What is PHP?* Library reference website, 2015k. Accessed on 27 May 2015.
URL `http://www.php.net/manual/en/intro-whatis.php`

Preda, M. and Giacobazzi, R. *Semantic-Based Code Obfuscation by Abstract Interpretation.* In **Caires, L., Italiano, G., Monteiro, L., Palamidessi, C., and Yung, M.**, editors, *Automata, Languages and Programming*, volume 3580 of *Lecture Notes in Computer Science*, pages 1325–1336. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-27580-0. doi:10.1007/11523468_107.

Python Software Foundation. *Subprocess Management.* Library reference website. Accessed: 27 October 2014.
URL `https://docs.python.org/2/library/subprocess.html`

Raber, J. and Laspe, E. *Deobfuscator: An automated approach to the identification and removal of code obfuscation.* In *wcre*, pages 275–276. IEEE, 2007. doi:10.1109/WCRE.2007.18.

Rajaram, S. and Oono, Y. *NeatMap - non-clustering heat map alternatives in R. BMC Bioinformatics*, 11(1):45, 2010. ISSN 1471-2105. doi:10.1186/1471-2105-11-45.

Roussev, V. *An Evaluation of Forensic Similarity Hashes. Digital Investigation*, 8:S34–S41, 2011.

Roussev, V. *Scalable data correlation.* In *Eighth annual IFIP Working Group*, volume 11. 2012.

Roussev, V. *Sdhash Home.* Package reference site, 2015. Accessed: 16 August 2015.
URL `http://roussev.net/sdhash/sdhash.html`

Roy, C. K. and Cordy, J. R. *NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization.* In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 172–181. IEEE, 2008.

Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Nieves, J., and Bringas, P. *MAMA: Manifest Analysis for Malware Detection in Android. Cybernetics and Systems*, 44:469–488, 2013.

Schroeder, W. J., Lorensen, B., and Martin, K. *The Visualization Toolkit.* Kitware, 2004. ISBN 193093419X.

Shankarapani, M. K., Ramamoorthy, S., Movva, R. S., and Mukkamala, S. *Malware Detection using Assembly and API Call Sequences. Journal in Computer Virology*, 7(2):107–119, 2011.

Sharif, M., Yegneswaran, V., Saidi, H., Porras, P., and Lee, W. *Eureka: A Framework for Enabling Static Malware Analysis.* In **Jajodia, S. and Lopez, J.**, editors, *Computer Security - ESORICS 2008*, volume 5283 of *Lecture Notes in Computer Science*, pages 481–500. Springer Berlin Heidelberg, 2008a. ISBN 978-3-540-88312-8. doi:10.1007/978-3-540-88313-5_31.

Sharif, M. I., Lanzi, A., Giffin, J. T., and Lee, W. *Impeding Malware Analysis Using Conditional Code Obfuscation.* In *Network and Distributed System Security.* 2008b.

Sklar, D. *Learning PHP 5.* O'Reilly Media, 2008. ISBN 9780596555351.

Slade, R. M. *Software Forensics. Tata McGrawHill*, 2004.

Sokal, R. R. and Rohlf, F. J. *The comparison of dendrograms by objective methods. Taxon*, 11:33–40, 1962.

Sun, H.-M., Lin, Y.-H., and Wu, M.-F. *API Monitoring System for Defeating Worms and Exploits in MS-Windows System.* In **Batten, L. and Safavi-Naini, R.**, editors, *Information Security and Privacy*, volume 4058 of *Lecture Notes in Computer Science*, pages 159–170. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-35458-1. doi:10.1007/11780656_14.

Suzumura, T., Trent, S., Tatsubori, M., Tozawa, A., and Onodera, T. *Performance Comparison of Web Service Engines in PHP, Java and C.* In *IEEE International Conference on Web Services*, pages 385–392. 2008. doi:10.1109/ICWS.2008.71.

Symantec Corporation. *Glossary of Security Terms.* 2015. Accessed: 14 July 2015.
URL `http://www.symantec.com/security_response/glossary/define.jsp?letter=t\&word=trojan-horse`

Tatroe, K. *Programming PHP.* O'Reilly & Associates Inc, 2005. ISBN 0596006810.

Terry, P. *Compiling with C# and Java.* Pearson Education, 2005. ISBN 032126360X.

The Resource Index Online Network. *The PHP Resource Index.* January 2005. Accessed on 24 May 2013.
URL `http://php.resourceindex.com/`

Titchkosky, L., Arlitt, M., and Williamson, C. *A performance comparison of dynamic Web technologies. SIGMETRICS Perform. Eval. Rev.*, 31(3):2–11, December 2003. ISSN 0163-5999. doi:10.1145/974036.974037.

Trent, S., Tatsubori, M., Suzumura, T., Tozawa, A., and Onodera, T. *Performance comparison of PHP and JSP as server-side scripting languages.* In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 164–182. Springer-Verlag New York, Inc., New York, NY, USA, 2008. ISBN 3-540-89855-7.

Ullman, L. *PHP for the world wide web: visual quickstart guide.* Peachpit Press, 2004.

United States Computer Emergency Readiness Team. *Compromised Web Servers and Web Shells - Threat Awareness and Guidance.* 2015. Accessed: 4 October 2015.
URL `https://www.us-cert.gov/ncas/alerts/TA15-314A`

Van Der Walt, S., Colbert, S. C., and Varoquaux, G. *The numpy array: a structure for efficient numerical computation. Computing in Science & Engineering*, 13(2):22–30, 2011.

VirusTotal Team. *VirusTotal Official Documentation.* Service reference website, 2015a. Accessed: 16 June 2015.
URL `https://www.virustotal.com/en/about/`

VirusTotal Team. *VirusTotal Private API.* Service reference website, 2015b. Accessed: 16 June 2015.
URL `https://www.virustotal.com/en/documentation/private-api/`

VirusTotal Team. *VirusTotal Public API.* Service reference website, 2015c. Accessed: 16 June 2015.
URL `https://www.virustotal.com/en/documentation/public-api/`

Walenstein, A. and Lakhotia, A. *The software similarity problem in malware analysis.* Internat. Begegnungs-und Forschungszentrum für Informatik, 2007. Accessed: February 2015.
URL `http://drops.dagstuhl.de/opus/volltexte/2007/964/`

Walenstein, A., Venable, M., Hayes, M., Thompson, C., and Lakhotia, A. *Exploiting similarity between variants to defeat malware.* In *Proc. BlackHat DC Conf.* 2007. Accessed: February 2015.
URL `https://www.blackhat.com/presentations/bh-dc-07/Walenstein/Paper/bh-dc-07-walenstein-WP.pdf`

Wang, C. *A security architecture for survivability mechanisms.* Ph.D. thesis, University of Virginia, 2001.

Wardman, B., Shukla, G., and Warner, G. *Identifying vulnerable websites by analysis of common strings in phishing URLs.* In *eCrime Researchers Summit, 2009. eCRIME'09.*, pages 1–13. IEEE, 2009.

Web Technology Surveys. *Usage statistics and market share of PHP for websites.* May 2013. Accessed on 24 May 2013.
URL `http://w3techs.com/technologies/details/pl-php/all/all`

Wilkinson, L. and Friendly, M. *The history of the cluster heat map. The American Statistician*, 63(2):179–184, 2009. doi:10.1198/tas.2009.0033.
URL `http://dx.doi.org/10.1198/tas.2009.0033`

Wrench, P. and Irwin, B. *Towards a Sandbox for the Deobfuscation and Dissection of PHP malware.* In *Information Security for South Africa (ISSA), 2014*, pages 1–8. Aug 2014. doi: 10.1109/ISSA.2014.6950504.

Wrench, P. and Irwin, B. *A Sandbox-based Approach to the Deobfuscation and Dissection of PHP-based Malware. South African Insitute of Electrical Engineers African Research Journal*, 106:46–63, 2015a.

Wrench, P. and Irwin, B. *Towards a PHP Webshell Taxonomy using Deobfuscation-assisted Similarity Analysis.* In *Information Security for South Africa (ISSA), 2015.* Aug 2015b. doi: 10.1109/ISSA.2014.6950504.

Wrench, P. and Irwin, B. *Detecting Derivative Malware Samples using Deobfuscation-assisted Similarity Analysis. South African Insitute of Electrical Engineers African Research Journal*, In Press, 2016.

Wu, A., Wang, H., and Wilkins, D. *Performance Comparison of Alternative Solutions For Web-To-Database Applications.* In *Proceedings of the Southern Conference on Computing*, pages 26–28. 2000.

You, I. and Yim, K. *Malware obfuscation techniques: A brief survey.* In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010. doi:10.1109/BWCCA.2010.85.

Yujian, L. and Bo, L. *A normalized Levenshtein distance metric. Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(6):1091–1095, 2007.

Zaremski, A. M. and Wing, J. M. *Signature matching: A key to reuse.* In *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '93, pages 182–190. ACM, New York, NY, USA, 1993a. ISBN 0-89791-625-5. doi:10.1145/256428.167077. URL `http://doi.acm.org/10.1145/256428.167077`

Zaremski, A. M. and Wing, J. M. *Signature matching: A key to reuse.* ACM, 1993b.

Zend Technologies. *The PHP Company.* February 2013. Accessed on 24 May 2013. URL `http://www.zend.com/en/resources/`

A

Antivirus Engines Aggregated by VirusTotal

AegisLab	Comodo	Malwarebytes Corporation
Agnitum	Doctor Web Ltd	McAfee Security
AhnLab	ESTsoft	Microsoft
Alibaba Group	Emsi Software GmbH	Microworld
Antiy Labs	Eset Software	Nano Security
ALWIL	Fortinet	Panda Security
Arcabit	FRISK Software	Qihoo 360
AVG Technologies	F-Secure	Rising Antivirus
Avira	G DATA Software	Sophos
BluePex	Hacksoft	SUPERAntiSpyware
Baidu	Hauri	Symantec Corporation
BitDefender GmbH	Ikarus Software	Tencent
Bkav Corporation	INCA Internet	ThreatTrack Security
ByteHero Information Security	Jiangmin	TotalDefense
Cat Computer Services	K7 Computing	Trend Micro
CMC InfoSec	Kaspersky Lab	VirusBlokAda
Cyren	Kingsoft	Zillya
ClamAV	Lavasoft	Zoner Software

B

Modules developed for the Viper Malware Analysis Framework

Module	**Description**
`Decode.py`	Reveals code hidden by `eval()` or `preg_replace()` constructs
`DecodeAll.py`	Decodes and normalises all samples
`Dendrogram.py`	Creates a dendrogram representation of a specified similarity matrix
`Download.py`	Submits and stores the results of malware retrieval requests
`Fetch.py`	Submits and stores the results of malware search queries
`FunctionBodies.py`	Extracts user-defined function bodies from a sample
`FunctionBodiesAll.py`	Extracts user-defined function bodies from all samples
`Functions.py`	Detects and extracts function names from a sample
`FunctionsAll.py`	Detects and extracts function names from all samples
`HashChunks.py`	Normalises, splits, and hashes a sample
`HashChunksAll.py`	Normalises, splits, and hashes all samples
`Heatmap.py`	Creates a heatmap representation of a specified similarity matrix
`HtmlDump.py`	Dumps the HTML generated by a sample
`HtmlDumpAll.py`	Dumps the HTML generated by all samples
`Matrix.py`	Creates similarity matrices based on a given measure of similarity

C

Code Availability

Although segments of code from the components developed for this research were included where necessary for the purposes of demonstration, the full listings were omitted brevity's sake. Copies of these listings are available online at https://github.com/Kalliades/Similarity-Analysis/tree/PHP/modules.