Towards a PHP Webshell Taxonomy using Deobfuscation-assisted Similarity Analysis

Peter M. Wrench* and Barry V. W. Irwin*
*Department of Computer Science, Rhodes University
Email: pete.wrench@gmail.com, b.irwin@ru.ac.za

***Abstract*—The abundance of PHP-based Remote Access Trojans (or web shells) found in the wild has led malware researchers to develop systems capable of tracking and analysing these shells. In the past, such shells were ably classified 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. This paper presents an approach to shell classification and analysis (based on similarity to a body of known malware) in an attempt to create a comprehensive taxonomy of PHP-based web shells. Several different measures of similarity were used in conjunction with clustering algorithms and visualisation techniques in order to achieve this. Furthermore, an auxiliary component capable of syntactically deobfuscating PHP code is described. This was employed to reverse idiomatic obfuscation constructs used by software authors. It was found that this deobfuscation dramatically increased the observed levels of similarity by exposing additional code for analysis.**

I. Introduction

PHP's popularity as a hosting platform [1] has made it the language of choice for developers of Remote Access Trojans (RATs) and other malicious software [2]. This software is typically used to compromise and monetise web platforms, 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, perform Distributed Denial of Service (DDOS) attacks, or serve as anonymous platforms for sending spam or other malfeasance [3].

Although many new shells are created every day, truly unique samples are rare - the vast majority of new threats are at least partially derivative, incorporating large portions of code from more established shells [4]. These subtle differences are often the result of malware authors adding functionality or attempting to make shells more resistant to signature-based matching techniques through the use of obfuscation. By investigating idiomatic deobfuscation techniques and different measures of similarity, this paper presents an alternative approach to malware analysis, with the goal of eventually developing a comprehensive taxonomy of web shells. Reference is made throughout the paper to work already published by the author in the area of code deobfuscation and normalisation [5].

This paper begins with an outline of a typical web shell and its common capabilities. The concept of code obfuscation is also introduced, with particular emphasis on how it is typically achieved in PHP. Section III also describes the ssdeep fuzzy hashing tool and its usefulness as a basis for similarity analysis and introduces Viper, the extendable malware framework that was used to store and manipulate malware samples. Section IV details how the system was designed and implemented, outlining both the deobfuscation process and the construction of similarity matrices and visual representations of sample similarity. The results obtained during system testing are presented in Section V. Finally, Section VI concludes the paper before Section VII presents ideas for future work and improvement.

II. Background and Related Work

This section begins by detailing research already carried out by the author into the creation of a module capable of syntactically deobfuscating PHP code [5]. This includes a description of the structure and capabilities of typical web shells and an overview of idiomatic code obfuscation techniques. The latter part of the section introduces the concept of code similarity and the various methods of testing for it, with particular emphasis on context-triggered piecewise hashing algorithms. The section concludes with a description of Viper, the static malware analysis framework that was used to store and manipulate the set of sample shells.

A. Web Shells

Remote Access Trojans (or web shells) are small scripts designed to be uploaded onto production servers. Once infected, a remote operator is able to control the server as if they had physical access to it [6]. Most web shells include features such as access to the local file system, keystroke logging, registry editing, and packet sniffing capabilities [3].

B. Code Obfuscation and PHP

Code obfuscation is a program transformation intended to thwart reverse engineering attempts [5]. Collberg et al. [7] define a code obfuscation as a "potent transformation that preserves the observable behaviour of programs". Although often used to protect proprietary code, code obfuscation is also employed by malware authors to hide their malicious code.

Reverse engineering obfuscated malware is a non-trivial, as the obfuscation process complicates the instruction sequences, disrupts the control flow and makes the algorithms difficult to understand.

As a procedural language with object-oriented features, PHP can be obfuscated using all of these methods. Additionally, several functions exist that directly support the hiding of code and which are often combined to form the following obfuscation idiom [5]:

```
eval(gzinflate(base64_decode('GISJg+S3Lrv...')));
```

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.

C. *Fuzzy Hashing and Ssdeep*

Hashing is a technique commonly used in forensic analysis that transforms an input string of arbitrary length into a fixed-length signature [8]. Once generated, these signatures can then be used to efficiently match identical files. Traditional hashing algorithms such as MD5 and SHA256 are designed such that changing just one bit in the input file will lead 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). Also known as fuzzy hashing, this technique combines piecewise hashing and rolling hashes to create a hash that is composed of values that only depend on part of the input. Piecewise hashing is the process of breaking an input into chunks and hashing these chunks separately, which means that changing part of the input file will only affect part of the resulting hash [9]. Because of this property, CTPH can be used to identify similar files as well as identical files. The rolling hash is used to provide the trigger points for separating the input into chunks by monitoring the context, which in this case is represented by the last n characters in a file [8].

Ssdeep is a hashing tool that was developed by Jesse Kornblum in 2006 [8]. 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 intensive than other algorithms such as MD5, but it is a far more effective way of identifying code reuse in similar files.

D. *Viper*

Viper [10] 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 - additional functionality can be added by simply creating a new module. It is this extensibility that prompted Viper's use as a basis for this project.

Access to a specific malware sample in Viper is achieved by opening a Viper session, 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, but any module can access multiple files by retrieving them from the database.

Malware samples in Viper can be organised into separate projects. 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 that is currently open.

III. Design and Implementation

This section begins by describing the Decode.py script, which is responsible for code deobfuscation and normalisation prior to analysis. The script's primary `decode()` function is also outlined, along with its two auxiliary functions, `processEvals()` and `processPregReplace()`. Four individual preprocessing modules are then introduced, each of which represent a unique measure of similarity. A brief description of the batch modules and their respective configurations is provided, as well as an overview of the Matrix.py module which is responsible for the creation of similarity matrices. Finally, the visualisation modules that are used to interpret and display these matrices are described in Section III-E.

A. *The Decode Script*

The purpose of the Decode.py script is to reveal the source code of a malware sample by removing any layers of obfuscation added by the author. To do this it makes use of the `decode()` function, which is described in Section III-A1. The two supporting functions, `processEvals()` and `processPregReplace()`, are described in Sections III-A2 and III-A3 respectively.

1) Decode: The purpose of the `decode()` function is to locate and process the `eval()` and `preg_replace()` constructs that can be used to execute arbitrary PHP code. The `eval()` functions executes any string argument as PHP code, and `preg_replace()` is able to execute the result of its search and replace in the same way. The `eval()` function in particular is often combined with auxiliary string manipulation functions to strengthen the obfuscation. The full pseudo-code of the `decode()` function is presented in Listing 1.

2) ProcessEvals: The `eval()` function can be used to execute an arbitrary string as PHP code, and as such is widely used as a method of obfuscation. 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

```
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 pretty printing
  Store the decoded shell in the database
END
```

Listing 1. Psuedo-code for the `decode()` function

situations, and that user-supplied data be strictly validated before being passed to the function. [11]

Listing 2 shows the full pseudo-code of the `processEvals()` function. This function is tasked with detecting `eval()` constructs in a script and replacing them with the code that they represent. 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 and each function is applied to the argument. The result of this process is then re-inserted into the shell in place of the original construct.

```
BEGIN
  WHILE there is still an eval in the script
    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

    Insert the resulting code
END
```

Listing 2. Psuedo-code for the `processEvals()` function

3) ProcessPregReplace: The `preg_replace()` function is used to perform a regular expression search and replace in PHP [12]. The danger of the function lies in the use of the deprecated ’/e’ modifier. If this modifier is included at the end of the search pattern, the interpreter will perform the replacement and then evaluate the result as PHP code, but the system prevents this from happening, as is demonstrated in Listing 3.

Listing 3 shows the full pseudo-code of the `processPregReplace()` function. It is tasked with detecting `preg_replace()` calls in a script and replacing them with the code 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 then 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 performed and its result can be inserted back into the script.

```
BEGIN
  WHILE there is still a preg_replace
    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 WHILE
END
```

Listing 3. Psuedo-code for the `processPregReplace()` function

B. The Indivdual 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 II-D for more information on sessions in Viper). Both Decode.py and HashChunks.py process samples in their entirety and produce an new file, whereas Functions.py and FunctionBodies.py extract relevant features for analysis.

1) Decode.py: The purpose of the Decode.py module is to remove idiomatic PHP obfuscation constructs from a single sample, thereby exposing more code for analysis and processing by the other three individual modules (all of which can be run on either raw or decoded samples for the purposes of comparison). It does this by accessing the Viper session, retrieving the open file, and passing it to the Decode.php script, the details of which are described in Section III-A. Once the script has reached completion, the resulting code is stored alongside the original script in the Viper repository.

2) 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 was the case with Decode.py and it’s accompanying Decode.php script.

3) Functions.py: The Functions.py module is similar to the FunctionBodies.py module, but it extracts just the names of any user-defined functions and ignores their associated bodies. As was the case with the FunctionBodies.py module, the feature extraction process is performed by an external PHP script, and the results are stored alongside the original file.

4) 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 II-C for more information on fuzzy hashing). The resulting hashes are then stored alongside the original file.

C. The 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 of the samples in the current project (see Section II-D 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 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 shown in Table I.

Module	Description
DecodeAll.py	Reveals hidden code for all samples
FunctionBodiesAll.py	Extracts function bodies from all samples
FunctionsAll.py	Creates a list of functions for all samples
HashChunksAll.py	Chunks and hashes all samples

TABLE I
THE BATCH MODULES AND THEIR DESCRIPTIONS

D. The 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 measure of similarity. It 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).

Several options can be passed to the matrix module. Each option represents the measure of similarity that should be used to generate a similarity matrix. If one would like to view the number of user-defined function name matches 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 shown in Table II.

Options	Description
-r	Compares raw samples using ssdeep
-d	Compares decoded samples using ssdeep
-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
-l raw	Compares the hashed chunks of raw samples
-l decoded	Compares the hashed chunks of decoded samples

TABLE II
THE POSSIBLE OPTION COMBINATIONS FOR MATRIX.PY

Each option (or measure of similarity) in the Matrix.py module is associated with a validation function and a comparison function. The validation function ensures that the batch functions needed to create the required files have been run successfully, and the comparison function calculates the observed similarity between two given files. A completed matrix represents the collation of the results returned by the comparison function for every pair of samples in the project.

E. The Visualisation Modules

The purpose of the visualisation modules is to create a graphical representation of a given similarity matrix. These representations are easier to interpret, and can be studied to discover relationships between samples.

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 II. Clusters of dark colours represent areas of greater similarity, while lighter areas indicate a lack of similarity.

2) Dendrogram.py: 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 measu res of similarity listed in Table II. 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.

IV. RESULTS

This section begins with a description of the collection of samples that was used for testing purposes. It then goes on to evaluate the effectiveness of the Decoder.py module and its attempts to normalise and deobfuscate samples prior to similarity analysis. A case study involving the c99 family of shells is then presented to demonstrate the results of the aforementioned analysis.

A. Test Data

During the testing process, 160 web shells were used as inputs to the system. These shells were primarily sourced from a comprehensive web malware collection maintained by Insecurety Research[1], which contains a variety of bots, backdoors and other malicious scripts. This collection was augmented with samples from other online sources. A breakdown of these sources is shown in Table III.

Source	Number of Shells
Insecurety.net	87
c99shell.gen.tr	21
r57shell.net	7
r57.gen.tr	10
hoco.cc	35

TABLE III
SAMPLE SOURCE BREAKDOWN

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

File sizes among the 160 shell samples ranged from 1.1kb to 546kb. An MD5 hash was generated for each file and compared to the hashes of every other file to ensure that no two files were identical. This process was repeated using SHA256, and both hashing algorithms were run both before and after deobfuscation for the sake of thoroughness. This was further reinforced during the comparison of the fuzzy hashes - 100% similarity was only ever observed when a shell was compared against itself.

B. Decode.py Tests

The decoder is responsible for performing code normalisation and deobfuscation prior to execution in the sandbox, with the goal of exposing the program logic of a shell. 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 the decoder. Finally, a test was performed with the fully-functional web shell as the input.

1) 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 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 string.

```
<?php
  eval(gzinflate(base64_decode(str_rot13('GIKK
    PhmVSslK+7V2LJg+S3Lrv...'))));
?>
```

Listing 4. Extract of a 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 5. 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 string manipulation functions, all with the same degree of success.

2) Full Shell Test: 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 is wrapped within 13 `eval(gzinflate(base64_decode()))` constructs, the outermost of which is partially displayed in Listing 6.

The decoder correctly produced the output shown in Listing 7. An analysis of the output found that all `eval()` and `preg_replace()` constructs had been correctly removed from the input script.

```
<?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 5. Extract of the expected decoder output with the script in Listing 4 as input

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

Listing 6. Extract of the outermost obfuscation layer as input

C. 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 results produced by the visualisation modules with a smaller subset of samples. The collection of samples used in this research contained seven variants of the popular c99 shell, which are listed below:

1) c99.txt
2) c99-bd.txt
3) c99_locus7s
4) c99madshell_v2.0.php
5) c99madshell_v2.1.php
6) c99shell_v1.0.php
7) c99ud.txt

For testing purposes, all of the option combinations were passed to the Matrix.py module in order to create all possible similarity matrices. These matrices were then processed by the visualisation modules to produce both heatmaps and dendrograms for every matrix. Sections IV-C1 and IV-C2 each demonstrate and analyse one example of each of these types of graph.

1) Heatmap.py Tests: The measure of similarity that was chosen to demonstrate the output produced by the Heatmap.py module was the user-defined function matching module (FunctionBodies.py) outlined in Section III-B2. The FunctionBodiesAll.py batch module was run against the family of c99 shells described in the previous section in both raw and decoded form, and the Matrix.py module was then used to create two similarity matrices based on the extracted function bodies. The matrix based on raw samples is shown in Figure 1, and the matrix based on decoded samples is shown in Figure 2. After running the Heatmap.py module against both matrices, the heatmaps shown in Figures 3 and 4 were produced. Darker colours represent a high level of similarity and vice versa.

Figure 3 reveals a relatively sparse distribution of similarity, with high values only occurring as a result of comparing samples against themselves. Of particular interest are the ud.txt and mad1.txt samples, which exhibit no similarity to the other shells in their raw forms. Clustering algorithms using this

```
<?php
  if(!function_exists("getmicrotime"))
  {
    functiongetmicrotime(){list($usec,$sec)...
  }
  error_reporting(5);
  @ignore_user_abort(TRUE);
  @set_magic_quotes_runtime(0);
  $win=strtolower(substr(PHP_OS,0,3))=="win";
  define("starttime",getmicrotime());
  ...
?>
```

Listing 7. Extract of the decoder output with the script in Listing 6 as input

	ud.txt	mad2.txt	v1.txt	c99.txt	locus.txt	bd.txt	mad1.txt
mad1.txt	0	0	0	0	0	0	100
bd.txt	0	14	37	10	25	100	0
locus.txt	0	9	17	12	100	25	0
c99.txt	0	4	8	100	12	10	0
v1.txt	0	8	100	8	17	37	0
mad2.txt	0	100	8	4	9	14	0
ud.txt	100	0	0	0	0	0	0

Fig. 1. Similarity matrix based on the function bodies extracted from raw c99 family shells

figure as an input would conclude that these two shells were not part of the c99 family of shells.

The similarity shown in Figure 4 differs slightly from that in Figure 3. In each case the values either increased or remained the same, which is to be expected when a larger portion of code is available for analysis. The decoded ud.txt and mad1.txt samples in particular demonstrated a far greater overall level of similarity to the rest of the collection. Upon examination of both the raw and decoded samples, it was discovered that these two shells were both encapsulated in `eval()` statements, which explains both their lack of similarity in Figure 3 and the subsequent increase shown in Figure 4.

2) Dendrogram.py Tests: The same measure of similarity (i.e. the comparison of extracted user-defined function bodies) was used to demonstrate the capabilities of the Dendrogram.py module so as to avoid the inclusion of two new matrices. Reference can therefore be made to the matrices depicted in Figures 1 and 2. The figures that were produced once the Dendrogram.py module had been run against these two matrices are shown in Figures 5 and 6 respectively.

The height of each cluster in a dendrogram represents the average distance between all inter-cluster pairs, and therefore the level of similarity between the samples that form that cluster. The lower the cluster height, the greater the similarity, and vice versa. As an example, consider the dendrogram shown in Figure 6. The c99.txt sample is more similar to

	ud.txt	mad2.txt	v1.txt	c99.txt	locus.txt	bd.txt	mad1.txt
mad1.txt	9	9	8	63	9	9	100
bd.txt	11	65	37	10	25	100	9
locus.txt	43	18	17	12	100	25	9
c99.txt	9	9	8	100	12	10	63
v1.txt	6	28	100	8	17	37	8
mad2.txt	10	100	28	9	18	65	9
ud.txt	100	10	6	9	43	11	9

Fig. 2. Similarity matrix based on the function bodies extracted from decoded c99 family shells

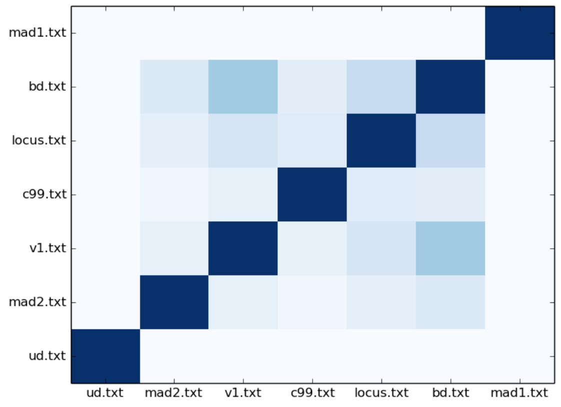

Fig. 3. Similarity heatmap based on the function bodies extracted from raw c99 family shells

mad1.txt than ud.txt is to locus.txt, because the first cluster is lower than the second. The two most similar samples are mad2.txt and bd.txt, because their cluster is the lowest on the dendrogram. These observations are supported by the values in the matrix shown in Figure 2, as the highest similarity value between two different shells is 65, which occurs between mad2.txt and bd.txt.

The difference between the similarity observed amongst raw and decoded samples is even more apparent from the change in the shape of the dendrogram from Figure 5 and Figure 6. The only pair of samples with any meaningful level of similarity in Figure 5 was observed between the v1.txt and bd.txt samples. As was the case with the heatmaps in Section IV-C1, all sample relationships either strengthened or remained the same.

V. Conclusion

The primary goal of this research was to determine the levels of similarity within a collection of malware samples.

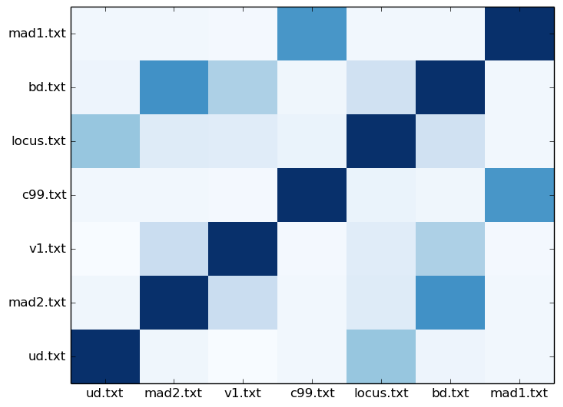

Fig. 4. Similarity heatmap based on the function bodies extracted from decoded c99 family shells

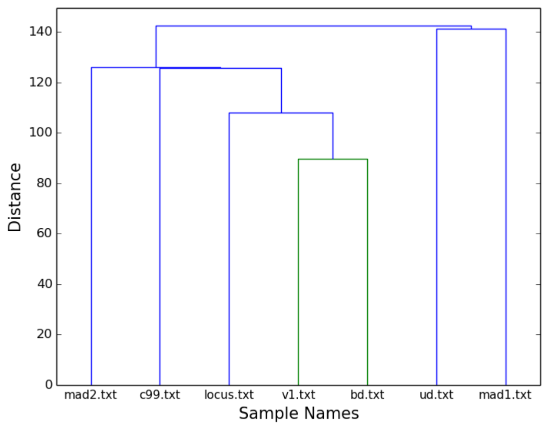

Fig. 5. Similarity dendrogram based on the function bodies extracted from raw c99 family shells

This was achieved by using four different measures of similarity to create representative similarity matrices, and then visualising and interpreting these matrices graphically. Section IV-C demonstrates the results of this process, and outlines how conclusions relating to sample similarity can be drawn by consulting either the matrices or their graphical representations. In addition to this, it was demonstrated that the deobfuscation process described in Section III-A was successfully able to increase the amount of code available for comparison, and thereby increase the accuracy of the similarity analysis process as a whole.

VI. Future Work

The development of different methods of similarity analysis and visualisation are intended to be used as a tools for creating detailed webshell taxonomies in the future. To this end, alternate methods of comparing shell samples need to be examined and other research into the evolution of malware needs to be investigated.

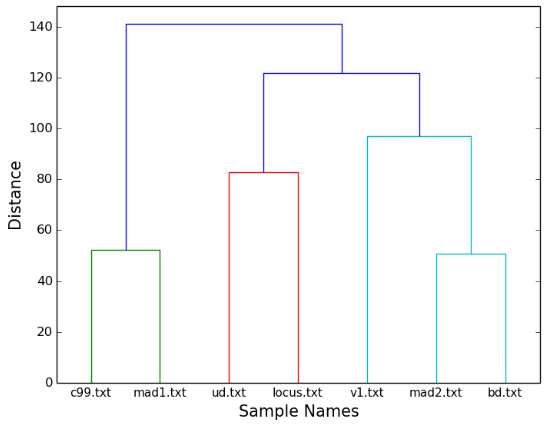

Fig. 6. Similarity dendrogram based on the function bodies extracted from decoded c99 family shells

A. *Alternative Shell Comparison Methods*

Although the four measures of similarity discussed in Section III-B are useful as measures of similarity, they represent only a few approaches to the detection of code reuse in webshells. 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 will be considered:

- HTML output matching
- Control graph matching
- Dynamic sandbox analysis
- Line-by-line analysis
- N-gram analysis
- Normalised compression distance

B. *A Webshell Taxonomy*

It is envisioned that this work will eventually lead to the construction of a taxonomy tracing the evolution of popular web shells such as c99, r57, b374k and barc0de [13] and their derivatives. This would involve the implementation of several tree-based structures that have the aforementioned shells as their roots and are able to show the mutation of the shells over time. Such a task would build on research into the evolutionary similarity of malware already undertaken by Li et al. [14], and would draw on the deobfuscation and similarity analysis capabilities described in this paper.

References

[1] K. Tatroe, *Programming PHP*. O'Reilly & Associates Inc, 2005.

[2] N. Cholakov, "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*, ser. CompSysTech '08. New York, NY, USA: ACM, 2008, pp. 12:II.7–12:2. [Online]. Available: http://doi.acm.org/10.1145/1500879.1500894

[3] M. Landesman. (2007, March) Malware Revolution: A Change in Target. Microsoft. Accessed on 1 March 2013. [Online]. Available: http://technet.microsoft.com/en-us/library/cc512596.aspx

[4] M. Doyle, *Beginning PHP 5.3*. Wiley, 2011. [Online]. Available: http://books.google.co.za/books?id=1TcK2bIJlZIC

[5] A. N. Other, “Towards a sandbox for the deobfuscation and dissection of PHP malware,” In press.

[6] R. Kazanciyan. (2012, December) Old Web Shells, New Tricks. Mandiant. Accessed on 1 March 2013. [Online]. Available: 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

[7] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[8] J. Kornblum. (2013, July) Context Triggered Piecewise Hashes. Accessed on 26 October 2013. [Online]. Available: http://ssdeep.sourceforge.net/

[9] L. Chen and G. Wang, “An efficient piecewise hashing method for computer forensics,” in *Knowledge Discovery and Data Mining, 2008. WKDD 2008. First International Workshop on*, Jan 2008, pp. 635–638.

[10] C. Guarnieri. (2014, March) Viper official documentation. [Online]. Available: http://viper-framework.readthedocs.org/en/latest/index.html

[11] The PHP Group. (2013, May) Eval. Accessed on 16 October 2013. [Online]. Available: http://php.net/manual/en/function.eval.php

[12] ——. (2013, May) Preg Replace. Accessed on 16 October 2013. [Online]. Available: http://php.net/manual/en/function.preg-replace.php

[13] T. Moore and R. Clayton, “Evil Searching: Compromise and Recompromise of Internet Hosts for Phishing,” in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, R. Dingledine and P. Golle, Eds. Springer Berlin Heidelberg, 2009, vol. 5628, pp. 256–272. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03549-4_16

[14] J. Li, J. Xu, M. Xu, H. Zhao, and N. Zheng, “Malware obfuscation measuring via evolutionary similarity,” in *First International Conference on Future Information Networks*, 2009, pp. 197–200.