DETECTING DERIVATIVE MALWARE SAMPLES USING DEOBFUSCATION-ASSISTED SIMILARITY ANALYSIS

P. Wrench* **and B. Irwin†**

* *Department of Computer Science, Rhodes University, Grahamstown, South Africa. Email: g10w1139@campus.ru.ac.za*

† *Department of Computer Science, Rhodes University, Grahamstown, South Africa. Email: 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.

Key words: Similarity Analysis, Code Hiding, PHP Malware, Remote Access Trojan

1. 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 frequently created, 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 authors in the area of code deobfuscation and normalisation [5, 6].

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 2 also describes the ssdeep fuzzy hashing tool and its usefulness as a basis for similarity analysis, and discusses the concept of data visualisation. Section 3 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 4. Section 5 concludes the paper before ideas for future work and improvement are presented in Section 6.

2. 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 (CTPH) algorithms. The section concludes by briefly describing two methods of visualising data similarity.

2.1 Web Shells

Remote Access Trojans (or web shells) are small scripts designed to be uploaded onto production servers. Once

Based on: "Towards a PHP Webshell Taxonomy using Deobfuscation-assisted Similarity Analysis", by P. Wrench and B. Irwin which appeared in the Proceedings of Information Security South African (ISSA) 2015, Johannesburg, 12 & 13 August 2015.

infected, a remote operator is able to control the server as if they had physical access to it [6, 7]. Most web shells include features such as access to the local file system, keystroke logging, registry editing, and packet sniffing capabilities [3].

2.2 Code Obfuscation and PHP

Code obfuscation is a program transformation intended to thwart reverse engineering attempts [5]. Collberg et al. [8] 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 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. Of the many built-in functions included in the core distribution of PHP, just two code execution functions account for the majority of code hiding efforts and are specifically marked by the PHP Group as being potentially exploitable [5, 9, 10].

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 [11].

The `eval()` function is often combined with auxiliary string manipulation functions to form the following obfuscation idiom [3]:

```
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 [12]. Although this doesn't 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:

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

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($a);`' and then evaluates the resulting code. In this case, the text contained in the `$a` variable would be displayed if the code was executed.

2.3 Lexical Analysis and the Zend Engine

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 [13]. 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 [14]. In PHP, lexical analysis is carried out by the Zend Engine, an open source interpreter originally developed by Andi Gutmans and Zeev Suraski [15].

The Tokenizer PHP extension [16] provides an interface to the lexical analyser used by the Zend Engine. 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 (see Section 3.3 for more detail on these processes).

2.4 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 [17]. Once generated, these signatures can then be used to efficiently match identical files. Traditional cryptographic hashing algorithms such as MD5 and SHA256 are designed so 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) [18]. 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 [19]. 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 [17].

Ssdeep is a hashing tool that was developed by Jesse Kornblum in 2006 [17]. 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.

2.5 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 [20]. Successful visualisation techniques should assist the viewer with analytical tasks such as making comparisons and identifying patterns in data. Although a wide variety of data visualisation structures exist, heatmaps and dendrograms are considered the most adept at highlighting similarity and relationships, and were thus selected as the tools for visualising the results of this research.

Heatmaps: Heatmaps are used to display each value in a given matrix as a colour that represents the magnitude of that value. Because of this property, the structures can be used to easily identify values (or areas of values) that represent a high level of similarity.

Dendrograms: Dendrograms are tree-like structures that can be used to display relationships that result from hierarchical clustering algorithms. 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.

3. DESIGN AND IMPLEMENTATION

This section begins by describing the decoder, which was developed and tested in previous research [5, 6] and 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()`, before Viper, (the malware analysis framework used in this research) is discussed. 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 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 3.6.

3.1 The 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 [21].

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, and the loss of confidential information. Static analysers are, however, unable to access runtime information (such as the value of a variable at a given point in the execution or the current program state) and are thus limited in terms of behavioral analysis.

The purpose of this component is to expose the underlying program logic of an uploaded 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 makes use of two core supporting functions, `processEvals()` and `processPregReplace()` and is described below.

Decode: The part of the decode script 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 PHP code. The `eval()` function interprets its string argument as PHP code, and `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.

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 get the output into a readable format before the decoded shell is stored in the database alongside its raw counterpart. The full pseudo-code of this process is presented in Listing 1.

ProcessEvals: The `eval()` function is able to evaluate an arbitrary string as PHP code, and as such is widely

```
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 1: Psuedo-code for the `decode()` function

used as a method of 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. [9]

As is described in Section 2.2, authors of malicious software often use the `eval()` function in conjunction with other string manipulation functions in order to further frustrate reverse engineering attempts. These functions typically compress, encode, or otherwise modify the string argument to increase the complexity of the obfuscation and thereby increase 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. A list of the functions that `processEvals()` is currently able to detect and process is shown in Table 1.

Function	Description
`base64_decode()`	Decodes data encoded by `base64_encode()`
`gzinflate()`	Inflates a deflated string `gzdeflate()`
`gzuncompress()`	Decompresses compressed data
`str_rot13()`	Restores a string encoded using `str_rot13()`
`strrev()`	Restores a string reversed using `strrev()`
`rawurldecode()`	Decodes data encoded using `rawurlencode()`
`stripslashes()`	Unescapes an escaped string
`trim()`	Strips whitespace from the edges of a string

Table 1: Auxiliary string manipulation functions that are handled by `processEvals()`

The `processEvals()` function was designed to be extensible. At its core is a switch statement that is 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 try and apply functions that it has not encountered before or been programmed to deal with.

Listing 2 shows the full pseudo-code of 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 then 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 2: Psuedo-code for the `processEvals()` function

ProcessPregReplace: The `preg_replace()` function is used to perform a regular expression search and replace in PHP [10].

```
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

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.

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. In order 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 is completed. A useful way of achieving this is to pass the script to a PHP parser which then creates an AST. 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 then be output according to a predefined set of formatting rules, ensuring that every sample conforms to the same formatting scheme.

Although the Zend engine that is used to interpret PHP can be used to split source code into an array of PHP tokens, it lacks the functionality to construct an AST and output it in a uniform way. For this reason, an open source lexical parser called PHP Parser was used to construct the AST and overwrite the existing sample text.

3.2 The Viper Framework

Viper [22] 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 research.

Projects: Malware samples in Viper can be organised into separate projects [23]. 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.

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.

Sessions: Access to a specific malware sample in Viper is achieved by opening a Viper session [24], 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 3.2 for information about 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 Table 3 for more information on commands in Viper). A summary of the information that each session object encapsulates is provided in Table 2.

Session Attribute	Description
current.file.path	The absolute path of the current file
current.file.name	The name of the current file
current.file.size	The size (in bytes) of the current file
current.file.type	The type and encoding of the current file
current.file.mime	The MIME type of the current file
current.file.md5	The MD5 hash of the current file
current.file.sha1	The SHA-1 hash of the current file
current.file.sha256	The SHA-256 hash of the current file
current.file.sha512	The SHA-512 hash of the current file
current.file.crc32	The CRC-32 check value for the current file
current.file.tags	A list of tags attached to the current file

Table 2: Attributes of a `__session__` object in Viper

The individual modules developed for this research (and described in Section 3.3) 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 2 in order to perform their respective tasks. An extract from the Decode.py module shown in Listing 4 demonstrates how the `is_set()` function of the global `__sessions__` object is used to check for the presence of an open session on line eight.

```
1  class Decode(Module):
2      cmd = 'decode'
3      description = 'Reveals hidden code'
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     ...
```

Listing 4: Extract from the Decode.py module demonstrating the use of the `is_set()` function

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 SQLLite database used to organise and store malware samples. Once imported, 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 3.4 all make use of the Database class's `find()` function to retrieve and process all samples in a given project. An extract from the Decode_All.py module shown in Listing 5 details how this is achieved on lines fourteen and fifteen.

```
1  from viper.core.database import Database
2
3  class Decode_All(Module):
4      cmd = 'decode_all'
5      description = 'Reveals hidden code'
6        in all samples'
7
8      def run(self):
9
10         # Get Viper's root path
11         viper_path = __project__.get_path()
12
13         # Retrieve all samples from the database
14         db = Database()
15         samples = db.find(key='all')
16
17         # Decode all samples
18         for sample in samples:
19             ...
```

Listing 5: Extract from the Decode_All.py module demonstrating the use of the `find()` function

Commands: Simple sample access and modification operations in Viper are carried out using commands [25]. 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 details all the available Viper commands and their respective uses.

3.3 The Indivdual Modules

Three 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 3.2 for more information on sessions in Viper). BDecode.py processes samples in their entirety and produces a new file, whereas Functions.py and FunctionBodies.py extract relevant features for analysis.

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 file notes 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 associated file tags to be modified

Table 3: Viper's core commands

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 3.1. Once the script has reached completion, the resulting code is stored alongside the original script in the Viper repository.

FunctionBodies.py: The purpose of the *FunctionBodies.py* script 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* script makes use of an external PHP script, as is the case with *Functions.py* and it's accompanying Functions.php script.

Functions.py: The purpose of the *Functions.py* script is to extract the names of any user-defined functions in a given sample. To do this, it makes use of PHP's Tokenizer extension to split a sample into tokens before looping through each token in search of the T_FUNCTION token type. Once this token type is found, the next string (representing the name of the function) is stored. Because the Tokenizer is implemented in PHP, an external PHP script called Functions.php was used to perform the name extraction process and return the results to the *Functions.py* script.

3.4 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 3.2 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 4.

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

Table 4: The batch modules and their descriptions

3.5 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 5.

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

Table 5: 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.

3.6 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.

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

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 5. 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.

4. 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.

4.1 Test Data

Although the malware samples used in this research were obtained from a variety of online sources (see Table 6 for a detailed source breakdown), the vast majority were retrieved from the collection of samples maintained by the VirusTotal online analysis service. VirusTotal allows researchers and commercial clients with access to a private 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.

The query string was 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

Source	Number of Shells
VirusTotal.com	1978
Insecurety.net	87
c99shell.gen.tr	21
r57shell.net	7
r57.gen.tr	10
hoco.cc	35

Table 6: Sample Source Breakdown

least one antivirus engine are included in the request. The full parameterised query is shown below:

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

File sizes among the 2138 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 was further reinforced during the comparison of the fuzzy hashes - 100% similarity was only ever observed when a shell was compared against itself.

4.2 *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. The results of these tests are omitted from this paper for the sake of brevity, but can be found in work previously published by the authors [5,6].

4.3 *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 samples used in this case study contain seven variants of the popular c99 shell, which are listed below:

1. c99.txt
2. c99-bd.txt
3. c99-locus.txt
4. c99-mad1.txt
5. c99-mad2.txt
6. c99-v1.txt
7. c99-ud.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.

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 3.3. 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.

	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

Figure 1: Similarity matrix based on the function bodies extracted from raw c99 family shells

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 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

	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

Figure 2: Similarity matrix based on the function bodies extracted from decoded c99 family shells

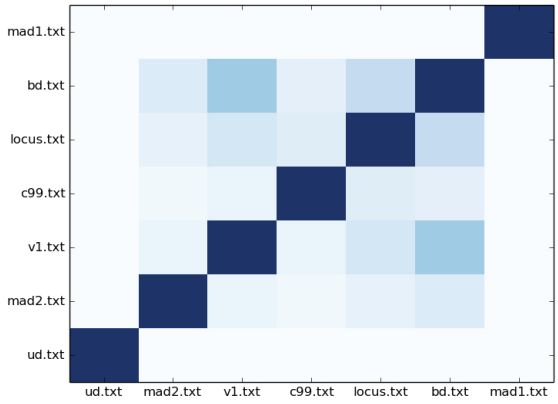

Figure 3: Similarity heatmap based on the function bodies extracted from raw c99 family shells

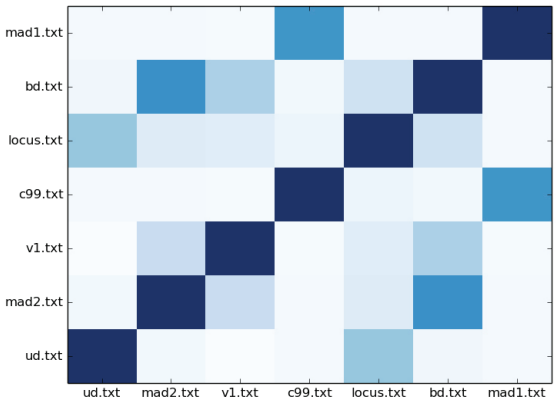

Figure 4: Similarity heatmap based on the function bodies extracted from decoded c99 family shells

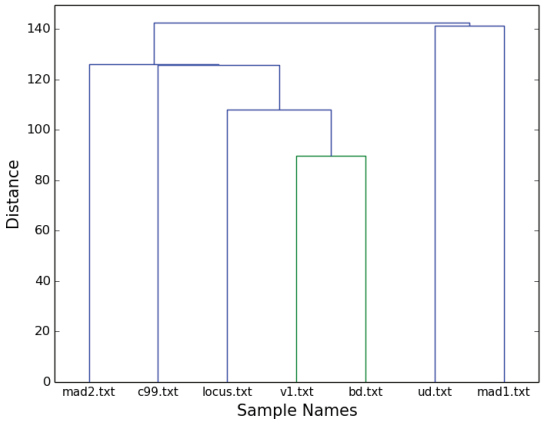

Figure 5: Similarity dendrogram based on the function bodies extracted from raw c99 family shells

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.

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 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

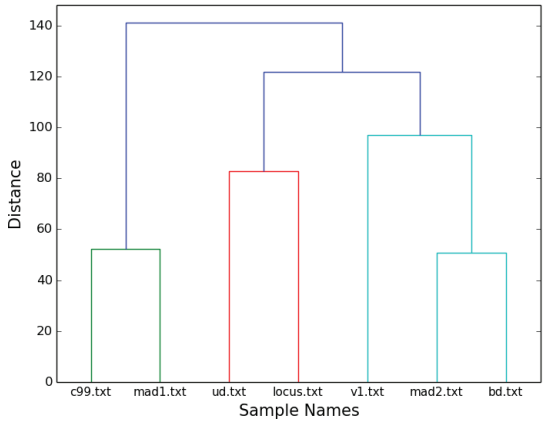

Figure 6: Similarity dendrogram based on the function bodies extracted from decoded c99 family shells

heatmaps in Section 4.3, all sample relationships either strengthened or remained the same.

4.4 Similarity Analysis Case Study: Cluster Identification

Although the smaller c99 case study is useful for demonstrating the similarity analysis process in a more concise and manageable way, the goal of the system is to identify areas of interest within a larger dataset. Once found, these areas could be subjected to further analysis. In order to demonstrate this process, 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 for this case study 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 upon by running the clustered samples through the decoder and then rendering the cluster again to gauge any differences in observed similarity.

Figure 7 shows the heatmap that was obtained by running the 150 shells through the analysis process. Once this was completed, an area of interest was selected for the purposes of demonstration. An enlarged version of this area is displayed in Figure 8. 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 is seen in Figure 9. A comparison of the heatmaps created before and after the deobfuscation process (shown in Figures 8 and 9 respectively) highlights the improvement in similarity due to the increased availability of code for analysis.

5. CONCLUSION

The primary goal of this research was to determine the patterns of similarity within a collection of malware samples. This was achieved by using four different measures of similarity to create representative similarity matrices, and then visualising and interpreting these matrices graphically. Section 4.3 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 3.1 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.

6. 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.

6.1 Alternative Shell Comparison Methods

Although the four measures of similarity discussed in Section 3.3 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 could be considered:

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

6.2 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 [26] 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. [27], and would draw on the deobfuscation and similarity analysis capabilities described in this paper.

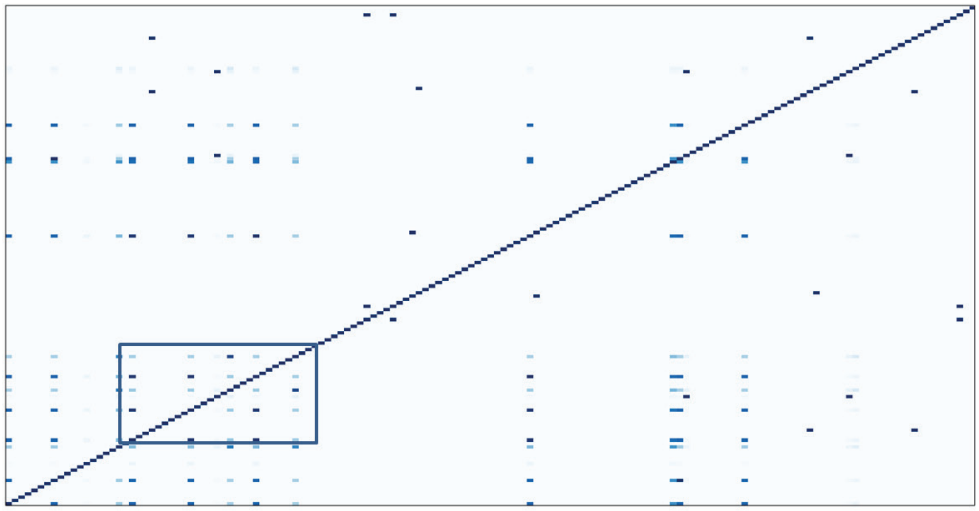

Figure 7: Similarity heatmap based on the function names extracted from a random selection of 150 raw shells

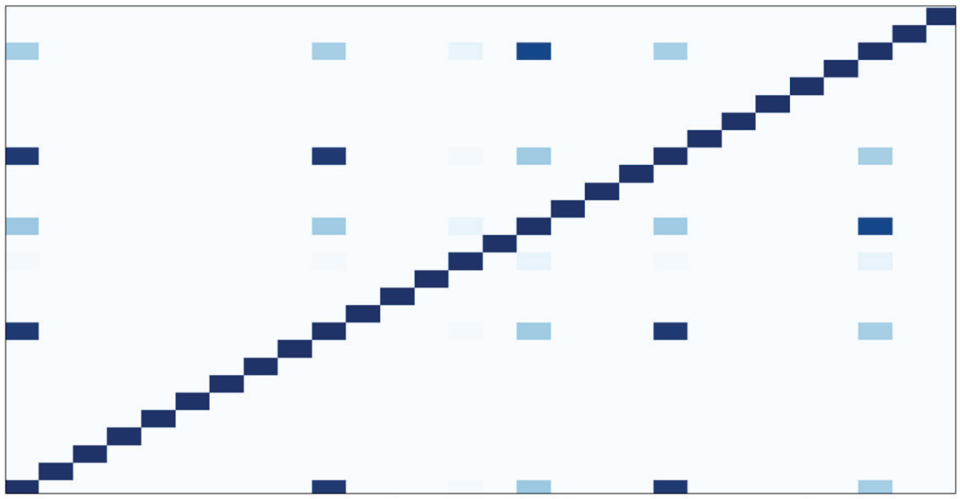

Figure 8: Focussed similarity heatmap based on the cluster identified in Figure 7

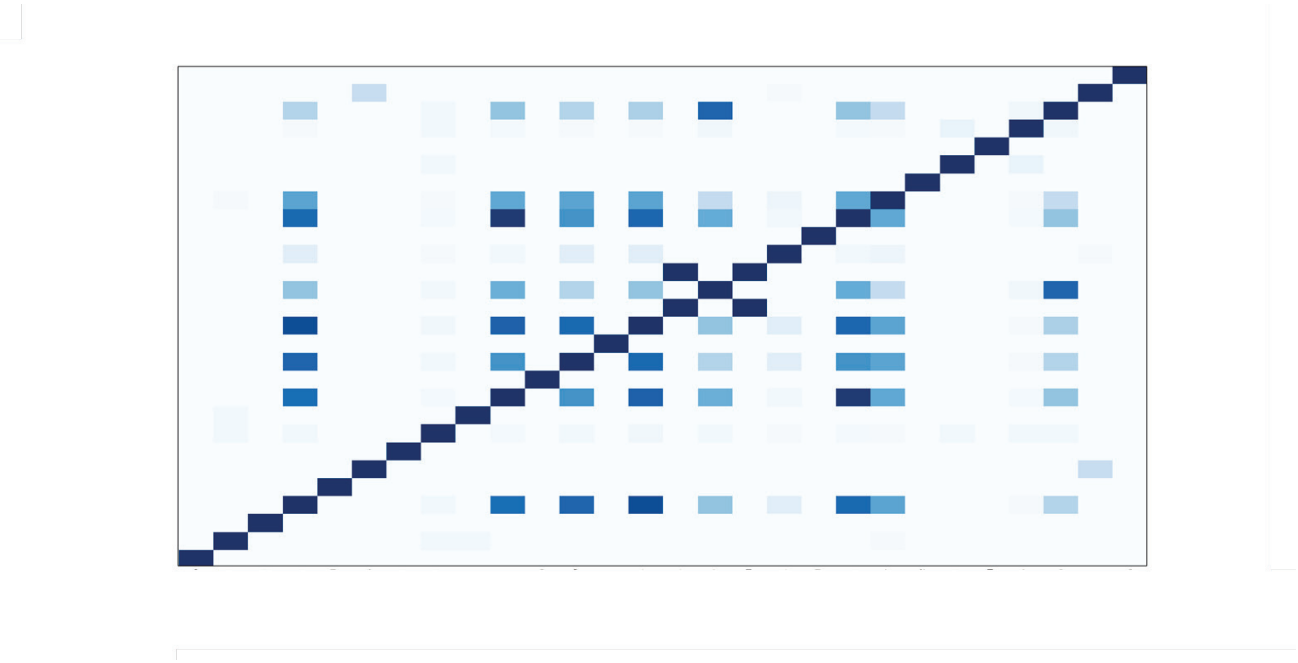

Figure 9: Similarity heatmap based on decoded version of the samples shown in Figure 8

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 *Information Security for South Africa (ISSA), 2014*. IEEE, 2014, pp. 1–8.

[6] ——, "A sandbox-based approach to the deobfuscation and dissection of php-based malware," *South African Insitute of Electrical Engineers*, vol. 106, pp. 46–63, 2015.

[7] 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

[8] 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.

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

[10] ——. (2013, May) Preg Replace. Accessed on 16 October 2013. [Online]. Available: http://php.net/manual/en/function.preg-replace.php

[11] A. Moser, C. Kruegel, and E. Kirda, "Limits of Static Analysis for Malware Detection," in *Twenty-Third Annual Computer Security Applications Conference*, December 2007, pp. 421–430.

[12] M. Christodorescu and S. Jha, "Testing malware detectors," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, pp. 34–44, Jul. 2004. [Online]. Available: http://doi.acm.org/10.1145/1013886.1007518

[13] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[14] P. Terry, *Compiling with C# and Java*. Pearson Education, 2005.

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

[16] The PHP Group. (2013, May) Tokenizer. Accessed on 16 October 2013. [Online]. Available: http://php.net/manual/en/intro.tokenizer.php

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

[18] ——, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3, Supplement, pp. 91 – 97, 2006, the Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287606000764

[19] 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.

[20] M. Friendly and D. J. Denis, "Milestones in the history of thematic cartography, statistical graphics, and data visualization," *http://www. datavis. ca/milestones*, 2001.

[21] M. Preda and R. Giacobazzi, "Semantic-Based Code Obfuscation by Abstract Interpretation," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, L. Caires, G. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, Eds. Springer Berlin Heidelberg, 2005, vol. 3580, pp. 1325–1336. [Online]. Available: http://dx.doi.org/10.1007/11523468_107

[22] C. Guarnieri. (2014, March) Viper official documentation. Accessed on 5 August 2015. [Online]. Available: http://viper-framework.readthedocs.org/en/latest/index.html

[23] ——. (2014, March) Viper projects. Accessed on 5 August 2015. [Online]. Available: http://viper-framework.readthedocs.org/en/latest/usage/concepts.html#projects

[24] ——. (2014, March) Viper sessions. Accessed on 5 August 2015. [Online]. Available: http://viper-framework.readthedocs.org/en/latest/usage/concepts.html#sessions

[25] ——. (2014, March) Viper commands. Accessed on 5 August 2015. [Online]. Available: http://viper-framework.readthedocs.org/en/latest/usage/commands.html

[26] 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

[27] 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.