

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

A Comparative Study of PHP Dialects

FINAL REPORT

Author:
Sher Ali KHAN

Supervisor:
Dr. Sergio MAFFEIS

June 16, 2015

Abstract

PHP is one of the most popular languages for server-side scripting. Because of its popularity, there are a number of different interpreters for it out there. Each one with a different performance and behaviour profile. We take a closer look at the four most popular ones, Zend, HHVM, HippyVM and Hack.

There is no place on the internet that allows a user to experiment with these dialects and explore the performance and behavioural differences between them.

To solve this problem, we present a web application that allows a user to securely execute snippets of PHP code on these four interpreters and run test suites that characterise the performance and behaviour of the interpreters. The behavioural test suite reveals strange and interesting characteristics about what actually happens under-the-hood that makes the interpreters behave so differently.

Acknowledgements

I would like to thank my supervisor, Dr. Sergio Maffei, for proposing the project and providing me with constant support throughout.

I would also like to thank my mother, for everything.

Contents

1	Introduction	8
1.1	Motivation	8
1.2	Objectives	9
2	Background	10
2.1	PHP	10
2.2	Zend	14
2.3	HipHop Virtual Machine	15
2.4	HippyVM	17
2.5	Facebook Hack	18
2.6	Other Interpreters	21
2.7	Security in PHP	22
2.7.1	Global Variables	22
2.7.2	Remote Files	24
2.7.3	Target Functions	25
2.8	Related Work	26
3	Technical Research	28
3.1	Jails	28
3.1.1	chroot Jail	30
3.1.2	FreeBSD Jail	31
3.2	Linux Containers (LXC)	32
3.3	Runkit_Sandbox	33
3.4	Conclusion	33
4	Design	34
4.1	Client-Side	34

4.1.1	Website Design	34
4.1.2	Test Suites	38
4.2	Server-Side	38
4.2.1	Web server	39
4.2.2	chroot Jail	39
4.3	Architecture	41
5	Implementation	43
5.1	Client-Side	43
5.2	Setting up Our chroot Jail	44
5.3	Process Router	45
5.4	Input Handler	46
5.4.1	Scanning Phase	46
5.4.2	Modifying Phase	49
5.5	Output Handler	51
6	Tests	52
6.1	Performance Tests	52
6.2	Behavioural Tests	57
7	Evaluation	68
7.1	The Web Application	68
7.2	Benchmarks and Test Suites	69
7.3	Limitations	70
7.3.1	Scaling Up	70
7.3.2	Security	70
8	Conclusions	72
8.1	Future Work	73
	Appendices	76
A		77

Chapter 1

Introduction

1.1 Motivation

In 1995, Rasmus Lerdorf created a personal collection of Perl scripts and transferred them into a package written in C. This package was called Personal Home Page tools, or PHP for short. Ten years later, PHP is used by 82% of the websites whose server-side programming language is known. The growth of PHP has been phenomenal; it has gone through several iterations over the years and now supports a wide range of complex features. While it was called a scripting language in the past, today it is more referred to as a dynamic programming language.

In the past, the Zend Engine was the only interpreter being used, but recently alternative, more efficient virtual machines for PHP have been proposed by Facebook (HHVM) and others (HippyVM). Facebook also recently introduced Hack, a PHP-like new language with gradual typing that also runs on the HHVM. Each implementation has a different performance profile, and changes the language behaviour more or less subtly.

There is no place on the internet that allows a user to experiment with the more popular PHP dialects and observe the performance and behavioural differences between them. This project aims to make a start towards filling that void.

1.2 Objectives

The goal of this project is to perform a comparative study of the PHP dialects used on these virtual machines, and produce a website reflecting the state of the art for the PHP language, as a much needed, impartial resource for the PHP developer community. The dialects being studied are Zend, HHVM, HippyVM and Facebook Hack. The website will provide an online interface to execute snippets of PHP code on the various interpreters.

Along with the website, we will identify language features that are peculiar only to some of these PHP dialects and use them to write a test suite that characterises the PHP constructs that behave differently across the implementations. We will also write a test suite that benchmarks the performance of the different interpreters.

These are interesting tasks because we need to look at what exactly is going on under the hood of these interpreters and identify the behavioural and performance differences in them. In addition to that, we also need to write test suites to show these differences to the users.

There are a number of beneficiaries who could make use of this website. It could help researchers in finding the differences in the interpreters or even companies deciding which dialect to use as their server-side web programming language. Provided that the websites front-end is not overwhelmingly technical, the website could also be used by more typical web users to educate them on the different dialects.

Chapter 2

Background

2.1 PHP

PHP, short for Hypertext Preprocessor, is an imperative server-side scripting language designed for web development but it is also used as a general-purpose programming language. It is used for developing complex programs and is used by not only amateur web developers but also billion-dollar companies such as Google, Facebook, Wikipedia and Yahoo!. PHP files end in with a .php extension.

PHP scripts are run on the server that hosts them, not on the clients machine. After the client makes a request for a web page using a browser, the request arrives at a web server. The web server loads the file required into memory. The web server sees that the file has a .php extension and therefore sends the file to the PHP interpreter which is also running on the web server. The PHP interpreter runs the PHP code in the file. At this point, dependent on the PHP code, the interpreter may further send a request to a database. When all the PHP code has finished running, the PHP interpreter sends the result back to the web server and the web server will then send the data to the browser.

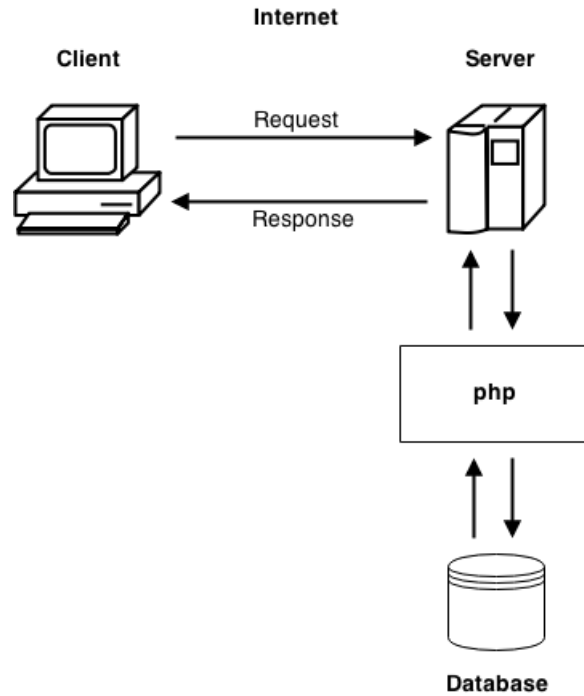


Figure 2.1: Client to PHP interaction.

PHP code can be embedded directly into HTML code. The ability to embed PHP code directly into HTML code is a huge advantage in creating simple but powerful websites and applications. It can also be used in combination with various templating engines and web frameworks.

PHP as a programming language was influenced by Perl, C, C++, Java and Tcl, and in terms of syntax, PHP is similar to most high level languages that follow the C style syntax. Listing 1 shows a very basic example of PHP code embedded into HTML code. The PHP interpreter only executes PHP code within the delimiters. Any code outside the delimiters is not processed by the PHP interpreter. The most common delimiters are `<?php` to open and `?>` to close PHP sections.

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>PHP Test</title>
5   </head>
6   <body>
7     <?php
8       echo "<p>Hello World</p>";
9     ?>
10  </body>
11 </html>
```

Listing 2.1: Hello world program written in PHP embedded in HTML.

```
1 <!DOCTYPE html>
2 <html>
3   <body>
4     <form method="post" action="<?php echo
5       $_SERVER['PHP_SELF'];?>">
6       Name: <input type="text" name="fname">
7       <input type="submit">
8     </form>
9
10    <?php
11      if ($_SERVER["REQUEST_METHOD"] == "POST") {
12        // collect value of input field
13        $name = $_POST['fname'];
14        if (empty($name)) {
15          echo "Name is empty";
16        } else {
17          echo $name;
18        }
19      }
20
21    </body>
22 </html>
```

Listing 2.2: PHP code using POST to collect form data after submitting an HTML form.

Listing 2.2 shows the use of superglobals and variables. Variables are prefixed with a dollar symbol and types do not need to be specified in advance. Superglobals were introduced in PHP 4.1.0, and are built-in variables that are always available in all scopes. The HTML form in listing 2.2 has an input field and a submit button. When the user submits the data by clicking Submit, the form data is sent to the file specified in the action attribute of the `<form>` tag. Then the super global variable `$_POST` is used to collect the value of the input field into a variable.

Apart from server-side scripting, PHP is most widely used for command-line scripting and with more advanced features, writing desktop applications. For command-line scripting, a PHP script can be made to run without any server or browser. These scripts can be used for regular text processing tasks or other complex tasks.

PHP can be used on all major operating systems and also has support for most of the web servers that exist today. This gives the freedom of choosing any operating system and almost any web server. Furthermore, PHP not only allows procedural programming but also object oriented programming. One of the strongest and most significant features in PHP is its support for a wide range of databases. Using a database specific extension or using an abstraction layer makes writing a database-enabled webpage incredibly simple. Because of these and many other features, PHP remains the most popular server-side scripting language. As of 18th February 2015, it is used by 82% of all websites whose server-side programming language is known.

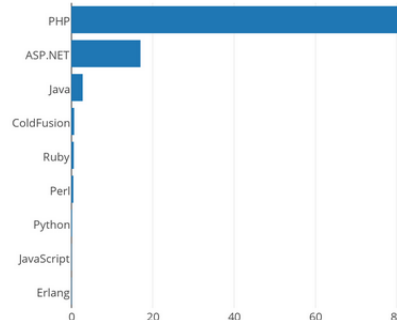


Figure 2.2: Comparison between different server-side languages.

2.2 Zend

The name Zend refers to the Zend Engine, which sits at PHP's core and acts as the runtime interpreter that compiles code in real time. The first version of the Zend Engine was a highly optimised modular back-end written in C that appeared in PHP 4.

Zend splits the processing of PHP code into several phases. The PHP code is run through a lexical analyser to convert the human-readable code into machine-digestible tokens. These tokens are then passed to the parser where the parser parses the tokens and generates a binary representation of the PHP code known as Zend Opcodes. Opcodes, short for operation codes, are low level binary instructions. Many parsers generate an abstract syntax tree or a parse tree before passing that to the code generator. However, the Zend Engine parser combines these steps and generates intermediate code directly from the tokens passed to it from the lexer. The opcodes are passed to an executor which generates HTML from the opcodes which can then be sent to the browser.

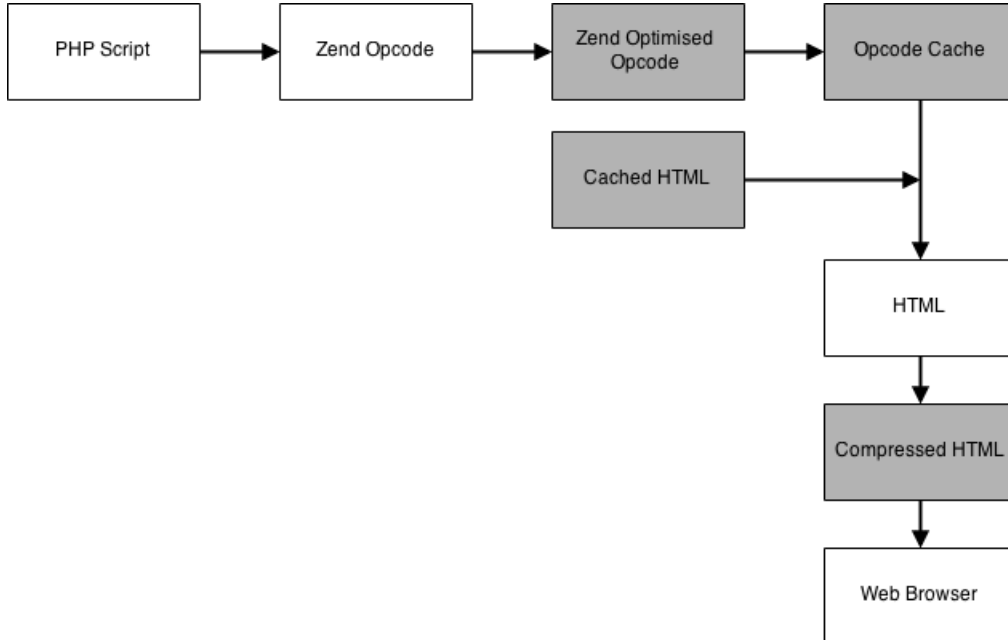


Figure 2.3: How the Zend Engine fits into PHP (gray steps are optional).

```

1 <?php
2     $hi = 'hello';
3     echo $hi;
4 ?>

```

	opnum	line	opcode	op1	op2	result
1	0	2	ZEND_FETCH_W	"hi"		'0
2	1	2	ZEND_ASSIGN	'0	"hello"	'0
3	2	3	ZEND_FETCH_R	"hi"		'2
4	3	3	ZEND_ECHO	'2		
5	4	5	ZEND_RETURN	1		

Listing 2.3: Representation of intermediate code for a simple PHP script.

Zend also provides memory and resource management for the PHP language. Zend is able to determine whether a block is in use, automatically freeing unused blocks and blocks with lost references, and thus prevents memory leaks. For resource management, Zend features a thread-safe resource manager to provide better native support for multi-threaded Web servers. The currently version at the heart of PHP 5 is The Zend Engine II.

2.3 HipHop Virtual Machine

HipHop Virtual Machine (HHVM) is an open-source virtual machine based on just-in-time (JIT) compilation, serving as an execution engine for the PHP and Hack programming languages. HHVM is developed by Facebook and is written in C and C++. It was created as the successor to HipHop for PHP (HPHPc) PHP execution engine, which is a PHP-to-C++ source-to-source compiler also created by Facebook. In early 2013, the production version of facebook.com switch from HPHPC to HHVM.

The executed PHP or Hack code is first parsed and analysed by HHVMs frontend and compiled into intermediate HipHop bytecode (HHBC). HHBC is a bytecode format created specifically for HHVM, in a form that is appropriate for consumption by interpreters and just-in-time compilers. The

HHBC code is then passed to HHVMs JIT compiler, interpreter and runtime where it is dynamically translated into x86-64 machine code, optimised and natively executed.

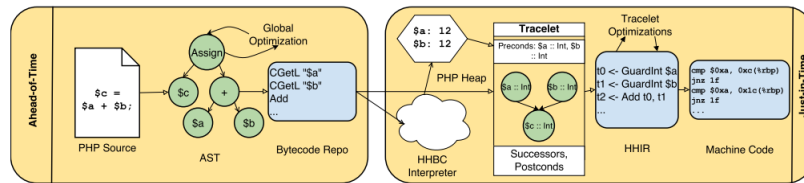


Figure 2.4: HHVMs production compilation pipeline.

The original motivation behind HipHop was to save resources on Facebook servers, given the large PHP codebase of facebook.com. As the development of HipHop progressed, it was realised that HipHop could substantially increase the speed of PHP applications in general. Increases in web page generation throughput by factors of up to six have been observed over the Zend PHP.

Due to performance gains and drops in the number of servers required for hosting, other websites such as wikipedia.org have also switched from Zend to HHVM. The performance gains Wikipedia has observed include the following:

- The CPU load on the servers has dropped drastically, from about 50% to 10%;
- The mean page save time has been reduced from $\tilde{6}$ seconds to $\tilde{3}$ seconds;
- The median page save time fell from $\tilde{7.5}$ seconds to $\tilde{2.5}$ seconds;
- The average page load time for logged-in users dropped from about 1.3 seconds to 0.9 seconds.

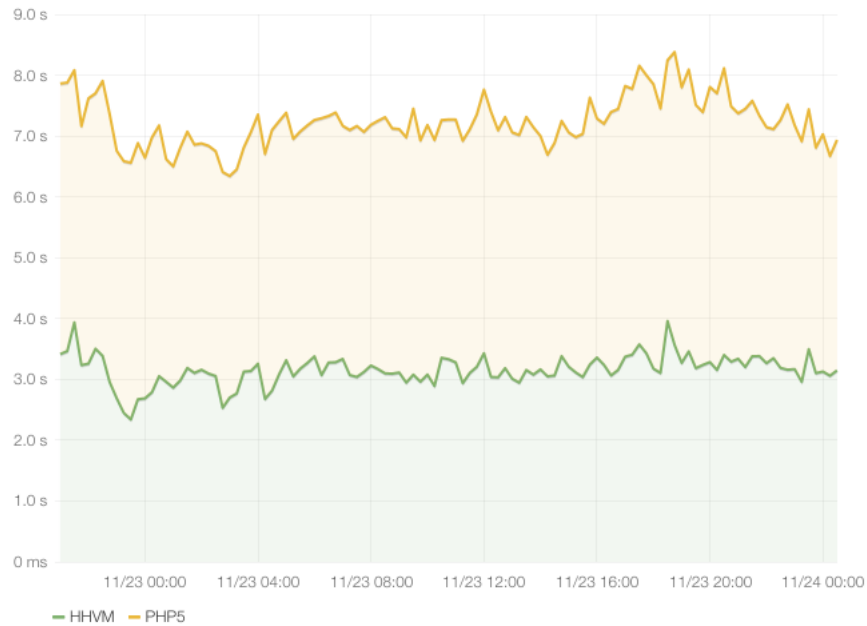


Figure 2.5: Median page save time for Wikipedia.

2.4 HippyVM

HippyVM is an implementation of the PHP language using RPython/PyPy technology. It started off as a Facebook-sponsored study on the feasibility of using the RPython toolchain to produce a PHP interpreter, and was later expanded upon.

HippyVM uses a tracing just-in-time (JIT) compiler to take HippyVM opcodes and generate the native machine code. Tracing just-in-time compilation is a technique used to optimise the execution of a program at runtime. This is done by recording a linear sequence of frequently executed operations, compiling them to native machine code and executing them. As opposed to traditional JIT compilers that work on a per-method basis. HippyVM is not complete yet but aims to be 100% compatible with Zend PHP and its proponents are claiming that it is faster than both Zend and HipHopVM.

Benchmark	Zend	HipHopVM	Hippy VM	Hippy/Zend	Hippy/HipHop
arr	2.771	0.508+-0%	0.274+-0%	10.1x	1.8x
fannkuch	21.239	7.248+-0%	1.377+-0%	15.4x	5.3x
heapsort	1.739	0.507+-0%	0.192+-0%	9.1x	2.6x
binarytrees	3.22	0.641+-0%	0.460+-0%	7.0x	1.4x
cachegetscb	3.350	0.614+-0%	0.267+-2%	12.6x	2.3x
řb	2.357	0.497+-0%	0.021+-0%	111.6x	23.5x
fasta	1.499	0.233+-4%	0.177+-0%	8.5x	1.3x

Figure 2.6: Performance difference between Zend, HipHopVM and HippyVM.

2.5 Facebook Hack

Hack is a programming language for the HipHop Virtual Machine (HHVM), created by Facebook as a dialect of PHP. The syntax Hack follows is almost the same but it adds several new features to help improve the quality of the code and take full advantage of HHVM to execute code faster. Hack allows programmers to use both dynamic typing and static typing, and it interoperates seamlessly with PHP. When mixing PHP and Hack code, the PHP code sections are not checked against Hacks rules, but regular PHP code can call Hack code and vice-versa because it is all part of the same program.

There is a type checking phase that verifies the consistency of code according to Hacks rules. The type checker is invoked before runtime to determine whether the code is typesafe.

Hack code is indicated through the `<?hh` marker at the top of a file, as opposed to the canonical `<?php` marker. And it is important to note that unlike PHP, Hack and HTML code do not mix.

Some of Hacks new features have been described below.

Generics. Hack introduces generics to PHP (in the same vein as statically type languages such as C# and Java). Generics allow classes and methods to be parameterised (a type associated when a class is instantiated or a method is called).

```
1 <?hh
2 class Box<T> {
3     protected T $data;
4
5     public function __construct(T $data) {
6         $this->data = $data;
7     }
8
9     public function getData(): T {
10        return $this->data;
11    }
12
13 }
```

Listing 2.4: Class with a generic type parameter T.

Collections. The PHP language provides one primary mechanism for expressing containers of elements: the PHP array. Hack adds container types and interfaces to PHP. Building on Hack’s support for generics, Hack adds first class, built-in parameterised collections such as vectors and maps. Collections are specialised for data storage and retrieval. Collections implement many of the same interfaces and are extendable to create even more specialized collections. Currently, Hack implements the following concrete collection types:

- Vector: An ordered, index-based list collection.
- ImmVector: An immutable, ordered, index-based list collection.
- Map: An ordered dictionary-style collection.
- ImmMap: An immutable, ordered dictionary-style collection.
- Set: A list-based collection that stores unique values.
- ImmSet: An immutable, list-based collection that stores unique values.
- Pair: An index-based collection that can hold exactly two elements.

```
1 <?hh
2
3 function main_col() {
4
5     $vector = Vector (5, 10);
6
7     $vector->add(15);
8     $vector->add(20);
9
10    $vector[] = 25;
11
12    $vector->removeKey(2);
13
14    foreach($vector as $item) {
15        echo $item . "\n";
16    }
17 }
18
19 main_col();
```

Listing 2.5: Example showing the use of the Vector collection in Hack.

Nullable. Hack introduces a safer way to deal with nulls through a concept known as the "Nullable" type. Nullable allows any type to have null assigned and checked on it. The ? operand is used to represent nullable.

```
1 <?hh
2 function check_not_null(?int $x): int {
3     if ($x === null) {
4         return -1;
5     } else {
6         return $x;
7     }
8 }
```

Listing 2.6: Example showing the use of the Nullable feature in Hack.

2.6 Other Interpreters

There are several other PHP interpreters that are continually being developed and expanded upon, but they are beyond the scope of this project. These other interpreters are minor in scale and do not have all the features that the current PHP version provides. Some of the interpreters do claim to be faster than the Zend engine but since they are not fully compatible with PHP, we will not be researching into them.

Recki Compiler Toolkit (Recki-CT)

Recki-CT is a compiler written entirely in PHP and only targets a subset of the PHP specification. It intentionally limits itself to a more static subset so that it is faster. This means that it does not support things like references, variable-variables and global variables.

Recki-CT compiles PHP down to machine code but unlike HHVM and HippyVM, which use Just in Time compilation to compile PHP, it uses Ahead of Time compilation which caches an intermediary representation that can be compiled at run-time. Therefore, more aggressive optimisations can be applied and more efficient code can be generated.

Based on trivial benchmarks, Recki-CT proves to be extremely fast. Somewhere between 10% to 15% faster than HHVM for average functions to somewhere in the order of 2 to 10 times faster than HHVM. These are artificial benchmarks and in production they would not be so significant, but this proves Recki-CT to be an interesting proof of concept.

phc

phc is an open source compiler for PHP with support for plugins. In addition, it can be used to pretty-print or obfuscate PHP code, as a framework for developing applications that process PHP scripts, or to convert PHP into XML and back, enabling processing of PHP scripts using XML tools. phc is no longer being developed but at its time, it was believed to be 1.5 times faster than the Zend engine.

2.7 Security in PHP

In this section, we describe some of the vulnerabilities in the PHP language and present some examples to showcase these vulnerabilities. Security is one of the key factors of our website since we will be executing user given PHP code on our servers. PHP offers a wide array of features, including running commands on the shell and making system calls, that can be used to attack a website such as ours.

2.7.1 Global Variables

Variables in PHP do not have to be declared and they are not specifically typed. They are automatically created the first time they are used and are automatically typed based on the context in which they are used. Once a variable is created it can be referenced anywhere in the program except in functions where it must be explicitly included in the namespace with the "global" function.

The main function of a PHP based web application, like ours, is usually to take in some user input in the form of variables, uploaded files, etc, process the input and return the desired output. A PHP script uses the global variables to access this input.

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>PHP Test</title>
5   </head>
6   <body>
7     <form method="GET" action="test.php">
8       <input type="TEXT" name="hello">
9       <input type="SUBMIT">
10    </form>
11  </body>
12 </html>
```

Listing 2.7: Use of PHP global variables.

The PHP code from listing 2.5 will display a text box and a submit button. When the user presses the submit button, the PHP script "test.php" will be run to process the input. When it runs the variable \$hello will contain the text the user entered into the text box. This means that a remote attacker can create any variable they wish and have it declared in the global namespace. If instead of using the form above to call "test.php", an attacker calls it directly with a url like "http://example.com/test.php?hello=hi&world=no", not only will \$hello be set to "hi" when the script is run, but also \$setup will be set to "no".

```
1 <?php
2     if ($pass == "hello")
3         $auth = 1;
4     ...
5     if ($auth == 1)
6         echo "Important Information";
7 ?>
```

Listing 2.8: Script that is designed to authenticate a user before displaying some important information.

In normal operation the code in listing 2.6 will check the password to decide if the remote user has successfully authenticated then later check if they are authenticated and show them the important information. The problem is that the code incorrectly assumes that the variable \$auth will be empty unless it sets it.

In normal operation the above code will check the password to decide if the remote user has successfully authenticated then later check if they are authenticated and show them the important information. The problem is that the code incorrectly assumes that the variable \$auth will be empty unless it is set. An attacker can create variables in the global namespace using a url like "http://example.com/test.php?auth=1" which will fail the password check but still successfully authenticate the attacker.

To summarize the above, a PHP script can not trust any variable it has not explicitly set.

2.7.2 Remote Files

PHP is an extremely feature rich language and ships with a vast amount of functionality out of the box. One of these features is the use of filesystem functions which would give the user the ability to read and write any file in our system.

```
1 <?php
2     if (!($fd = fopen("$filename", "r")))
3         echo("Could not open file: $filename<BR>\n");
4 ?>
```

Listing 2.9: PHP code designed to open a file.

The code in listing 2.7 attempts to open the file specified in the variable `$filename` for reading and if it fails displays an error. For example, a security issue could be if the user can set `$filename` and get the script to read from `/etc/passwd`. Another security issue with the same code is that PHP's file handling functions can work transparently on remote files via HTTP and FTP. If the variable `$filename` were to contain `"http://attack.com/script/cmd.exe/c+dir"`, PHP will actually make a HTTP request to the specified server for the file.

This gets more interesting in the context of four other file functions that support remote file functionality, `include()`, `require()`, `include_once()` and `require_once()`. These functions take in a filename and read that file and parse it as PHP code. They're typically used to support the concept of code libraries, where common bits of PHP code are stored in files and included as needed.

```
1 <?php
2     include($libdir . "/languages.php");
3 ?>
```

Listing 2.10: Use of the `include()` function.

```
1 <?php
2     passthru("/bin/ls /etc");
3 ?>
```

Listing 2.11: Attack code.

In listing 2.8 \$libdir is a configuration variable that is meant to be set earlier in script execution to the directory where the library files are stored. If the attacker can cause the variable not to be set in the script, they can modify the start of the path. This would normally gain them nothing since they still end up only being able to access languages.php in a directory of their choosing but with remote files the attacker can submit any code they wish to be executed. For example, if the attacker places a file on a web server called languages.php containing the code from listing 2.9 and then sets \$libdir to "http://evilhost.com/". Then upon encountering the include statement PHP will make a HTTP request to "evilhost", retrieve the attackers code and execute it, returning a listing of /etc to the attackers web browser.

2.7.3 Target Functions

PHP has a number of functions that are frequently misused or are good targets if they happen to be used in a vulnerable manner in the target application. If a remote attacker can affect the parameters of these functions, exploitation is often possible. The following is a non exhaustive breakdown.

PHP Code Execution:

- require() and include() – Both these functions read a specified file and interpret the contents as PHP code.
- eval() – Interprets a given string as PHP code and executes it.
- preg_replace() – When used with the /e modifier this function interprets the replacement string as PHP code.

Command Execution:

- `exec()` – Executes a specified command and returns the last line of the programs output.
- `passthru()` – Executes a specified command and returns all of the output directly to the remote browser.
- ```` (backticks) - Executes the specified command and returns all the output in an array.
- `system()` – Much the same as `passthru()` but does not handle binary data.
- `popen()` – Executes a specified command and connects its output or input stream to a PHP file descriptor.

File Disclosure:

- `fopen()` – Opens a file and associates it with a PHP file descriptor.
- `readfile()` – Reads a file and writes its contents directly to the remote browser.
- `file()` – Reads an entire file into an array.

2.8 Related Work

Research into related work has revealed that there is no other website that does what we have proposed. Users compare the advantages and disadvantages of the different PHP dialects on online blog posts and research papers but there is no website where snippets of code for the different dialects can be executed or where test suites characterising the dialects are available.

There are many websites which also users to execute snippets of PHP code in only the Zend engine. These websites also have certain security restrictions to keep their servers safe, such as blocking functions that allow executing shell commands and having a memory limit for the snippets of code.

The closest application to our project is 3v4l.org. It is an online shell where the user can execute snippets of PHP code and the site shows the output

and performance (in time and memory) from all released PHP and HHVM versions. It does a relatively good job of comparing Zend and HHVM but does not support HippyVM or Hack. It also has a few examples that show behavioural and performance differences in Zend and HHVM. Currently, this is the website that is being used by most of the people looking to execute snippets of PHP code online. The main reason for this website's popularity is that it has very few restrictions and also allows users to save and share code with each other. One of the requested features from 3v4l.org is the support for HippyVM which our website provides.

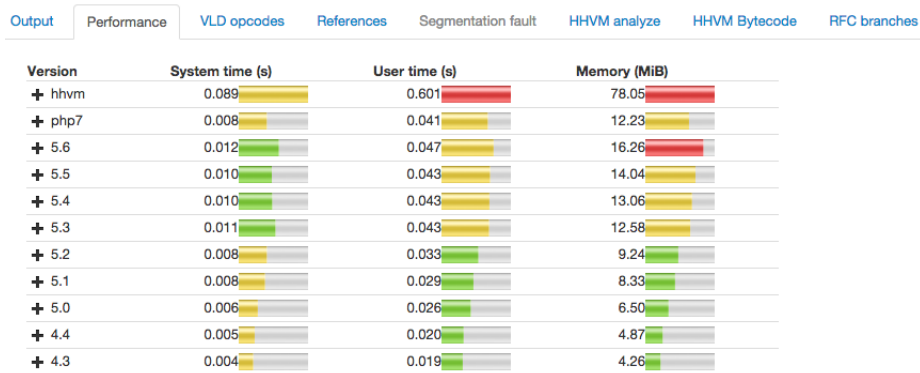


Figure 2.7: Performance result from running PHP code on 3v4l.org.

Chapter 3

Technical Research

For our website, we will be executing snippets of PHP code inputted by anyone on the internet. As mentioned earlier, PHP is a language with many vulnerabilities and allowing users to run code on our server drastically increases the risk of an attack. To overcome these security issues, we researched into creating a sandbox environment where the code can be executed.

A sandbox is a security mechanism for separating running programs. It is often used to execute untested code, or untrusted programs from unverified third parties, suppliers, untrusted users and untrusted websites. A sandbox typically provides a tightly controlled set of resources for programs to run in, such as scratch space on disk and memory, and limited access to the filesystem. Network access, the ability to inspect the host system or read from input devices are usually disallowed or heavily restricted.

We looked at a number of different solutions to provide the functionality we needed while still being compatible with all four dialects.

3.1 Jails

Jails are a fairly old concept in unix but they provide the sandbox environment we need. The jail puts the user into a restricted filesystem where they are unable to break out of. Inside the jail, the user can carry out activities just like a regular user but they do not have any affect on the real filesystem

outside of the jail. The user can only leave the jail with the help of the root user.

	running as root	running as non-root
entire filesystem	(a)	(c)
jailed filesystem	(b)	(d)

Figure 3.1: Categories in which a process may be running.

Using figure 3.1, we can illustrate the benefits of jailing:

- In category (a) (processes running as root with access to the entire filesystem). Compromising such a service would allow the attacker to replace binaries, to open privileged network ports, and read any file on the system. Worst, they have the capacity for complete damage.
- In category (b) (root processes running in a jail). A process in this state can break out of a jail. Given that the process is running as root, the attacker could use an exploit to execute code which makes system calls to perform root activities. Though much safer than state (a) in the context of a scripted attack, state (b) does not provide the strongest defence.
- In category (c) (non-root processes running with access to the entire filesystem). The threat for a full system compromise is reduced slightly from (a) in that the attacker will not immediately have root permissions. However, any process in this state can execute all the standard commands and shells, and thus allow the opportunity for an attacker to explore the filesystem in search of root-level exploits.
- In category (d) (non-root processes running in a jail). A compromised service would give an attacker no opportunity to execute shells or common commands or to explore the system information. Also, the extent of damage posed by file deletion is limited to directories with the jail.

We looked at the chroot and FreeBSD jails for a solution.

3.1.1 chroot Jail

A chroot (change root) operation changes the apparent root directory for a running process and its children. It allows the user to run a program with a root directory other than `/`. The program cannot see or access files outside the designated directory tree. Such an artificial root directory is called a chroot jail, and its purpose is to limit the directory access of a potential attacker. The chroot jail locks down a given process and any user ID that it is using so that all they see is the directory in which the process is running. To the process, it appears that the directory in which it is running is the root directory.

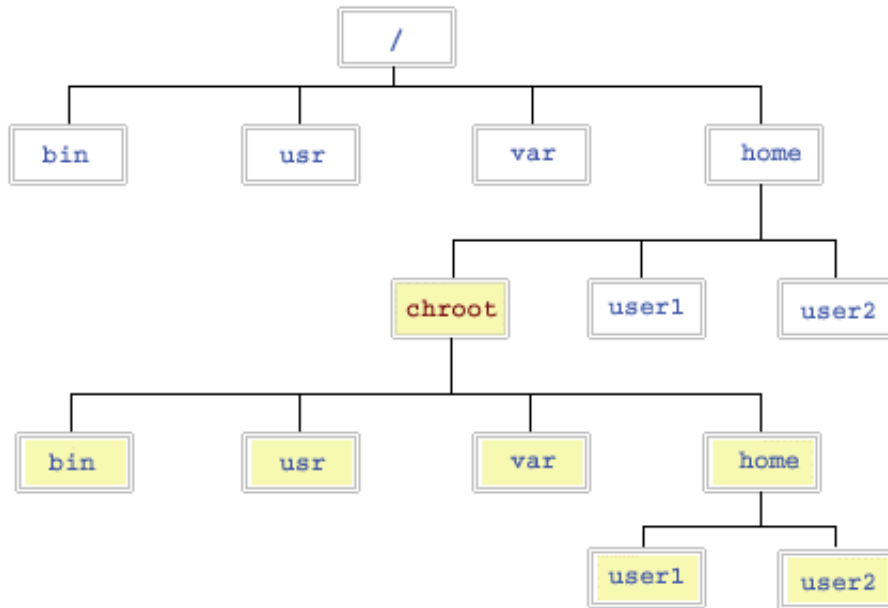


Figure 3.2: Directory tree showing the chroot jail.

A chroot environment can be used to create and host a separate virtualised copy of the software system. This can be useful for :

- **Testing/Development.** A test environment can be set up in the chroot for software that would otherwise be too risky to deploy on a production system.
- **Minimal Dependencies.** Software can be developed, built and tested in a chroot populated only with its expected dependencies. This can prevent some kinds of linkage skew that can result from developers building projects with different sets of program libraries installed.
- **Recovery.** Should a system be rendered unbootable, a chroot can be used to move back into the damaged environment after bootstrapping from an alternate root file system.
- **Compatibility.** Legacy software or software using a different application binary interface must sometimes be run in a chroot because their supporting libraries or data files may otherwise clash in name or linkage with those of the host system.
- **Privilege separation.** Programs are allowed to carry open file descriptors into the chroot, which can simplify jail design by making it unnecessary to leave working files inside the chroot directory. This also simplifies the common arrangement of running the potentially vulnerable parts of a privileged program in a sandbox, in order to pre-emptively contain a security breach.
- **Cross-Compilation.** Setting up a 32 bit chroot instead of a complete cross compilation toolchain allows a 64 bit system to compile for a 32 bit system.

Testing/Development and privilege separation are the characteristics of a chroot jail that will allow us to securely execute PHP code.

3.1.2 FreeBSD Jail

The FreeBSD jail is an implementation of operating system-level virtualisation that allows administrators to partition a FreeBSD-based computer system into several independent mini-systems called jails.

FreeBSD jails improve on the concept of the traditional chroot environment in several ways. In a traditional chroot environment, processes are only limited in the part of the file system they can access. The rest of the system resources, system users, running processes, and the networking subsystem are shared by the chrooted processes and the processes of the host system. FreeBSD jails expand this model by virtualising access to the file system, the set of users, and the networking subsystem. More fine-grained controls are available for tuning the access of a jailed environment.

FreeBSD jails are generally used when multiple jails are required within one system. FreeBSD jails partition the system and allow the administrators to manage and control multiple jails easily. Jails have their own set of users and their own root account which are limited to the jail environment. The root account of a jail is not allowed to perform operations to the system outside of the associated jail environment. Each jail is also assigned an IP address and a hostname.

3.2 Linux Containers (LXC)

The Linux Containers (LXC) feature is a lightweight virtualisation mechanism that does not require the user to set up a virtual machine on an emulation of physical hardware.

The Linux Containers feature takes the cgroups resource management facilities as its basis and adds POSIX file capabilities to implement process and network isolation. You can run a single application within a container (an application container) whose name space is isolated from the other processes on the system in a similar manner to a chroot jail. However, the main use of Linux Containers is to allow the user to run a complete copy of the Linux operating system in a container (a system container) without the overhead of running a level-2 hypervisor. In fact, the container is sharing the kernel with the host system, so its processes and file system are completely visible from the host. When the user is logged into the container, they only see its file system and process space. Because the kernel is shared, the user is limited to the modules and drivers that it has loaded.

3.3 Runkit_Sandbox

Runkit_Sandbox is a class from the runkit package inside the PHP language and instantiating it creates a new thread with its own scope and program stack. Using a set of options passed to the constructor, this environment may be restricted to a subset of what the primary interpreter can do and provide a safer environment for executing user supplied code.

```
1 <?php
2     $sandbox = new Runkit_Sandbox();
3
4     echo $sandbox->str_replace('a','f','abc');
5 ?>
```

Listing 3.1: Calling PHP functions using Runkit_Sandbox.

3.4 Conclusion

After researching into the different options, the best solution proved to be a chroot jail. The chroot jail was chosen because it was compatible with all four dialects and provided a minimal and secure system to execute PHP code.

Using a FreeBSD jail or a Linux Container does provide a secure system like a chroot jail but ultimately they are not minimal and cater to a wide range of features that we will not use. Our only use from a jail would be executing PHP scripts on the command line. The Runkit_Sandbox also creates a satisfactory virtual machine to run PHP code but it is not supported by HHVM and HippyVM. Using the Runkit_Sandbox would also mean using the eval function to run the PHP code which would drastically decrease the performance of the code.

Chapter 4

Design

Our web application consists of both a client-side application and a server-side application. The client-side runs in the users browser whilst the server-side runs on our web server. The two parts of our application work together in order to securely execute snippets of PHP code. Put simply, the PHP code from the client-side is sent to the server where it is executed and the output is sent to the client.

4.1 Client-Side

The client-side of our application is written in JavaScript, the only client side scripting language that is fully supported across all of todays browsers. It is responsible for running sending the PHP code from the user to the server-side application and getting the response (through AJAX).

4.1.1 Website Design

An important aspect to the project is the design of the website itself. The purpose of the website was a key factor when designing the website. Anyone visiting our website would obviously be looking to run their PHP code in the dialects of their choosing and see the results. The other key factor in our design was user experience. We want users to have a positive experience

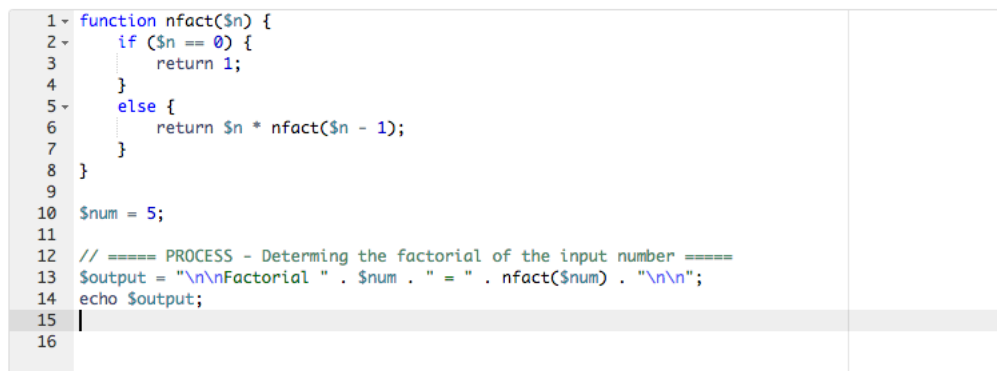
on our website so that may visit it again to run other snippets of PHP code.

Taking all this into account, we have produced a design that we believe to address the key factors mentioned earlier. It is based around Twitter's front-end framework *Bootstrap*¹ which makes it easy to produce a layout that works consistently across all browsers. *Bootstrap* also allowed us to produce a design that did not require us to spend too much time writing and testing our own CSS stylesheets and Javascript. We will now discuss the main components of our design that we believe are most of important to the website.

Editor

For our web application, we needed a multi-line text editor for the user to write their PHP code in. We could have simply used the HTML `<textarea>` tag to do so but we chose a more functional editor, Ace.

Ace is an embeddable code editor written in JavaScript. It is easily embedded in any web page and JavaScript application. It has a vast number of features, including syntax highlighting, highlighting matching parentheses and live syntax checking. It also has a great API reference manual which makes it easy to use.

A screenshot of the Ace code editor. The editor displays PHP code for calculating a factorial. The code is as follows:

```
1 function nfact($n) {
2     if ($n == 0) {
3         return 1;
4     }
5     else {
6         return $n * nfact($n - 1);
7     }
8 }
9
10 $num = 5;
11
12 // ===== PROCESS - Determining the factorial of the input number =====
13 $output = "\n\nFactorial " . $num . " = " . nfact($num) . "\n\n";
14 echo $output;
15
16
```

The code is syntax-highlighted, with keywords in blue, strings in green, and comments in grey. The editor has a light grey background and a dark grey border.

Figure 4.1: Ace editor being used on our website.

When users visit our website, the first thing they see is a large editor. They

¹<http://getbootstrap.com/>

can copy code into it or immediately start typing. The editor itself presents an extremely user friendly interface. It matches the style of native editors such as Sublime, Vim and TextMate so first time users will know exactly where to enter their code.

Choosing Dialects and Running

We allow our users to choose which interpreter they want to use when running the code. This is done in a very intuitive manner using checkboxes placed to the left of the *Run* button. When the *Run* button is pressed it locks, changes colours and changes to *Running...* until a response from the server is retrieved. The changes to the button lets the user know their PHP code is being executed.

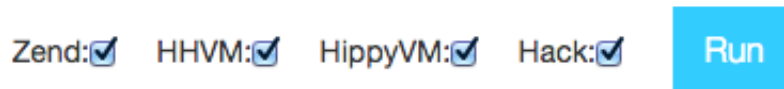


Figure 4.2: Choosing dialects and running on our website.

Other websites that offer multiple PHP interpreters do not allow their users to choose which dialects to run the code in. Their users have actually asked for this feature to be added as the aggregate time from running large snippets of code on all dialects can often take extremely long.

Also, other websites navigate away from their editor page and become static after their respective *Run* button is pressed. Our website stays on the same page, the editor remains active and the results are dynamically populated in their respective fields.

Results

The results are dynamically populated in their respective fields once a response is received from the server. The results include the output from executing the user given PHP code and the time it took to execute it. The result can also be the error messages produced from the PHP code if it failed to execute.

Zend: Hello, World!	0.00001699s
HHVM: Hello, World!	0.00056109s
HippyVM: Hello, World!	0.00001299s
Hack: Hello, World!	0.00056209s

Figure 4.3: Valid results for all four dialects.

If the PHP code is not allowed to execute because of our security checks, an appropriate message is shown. For example, a message could be shown telling the users which functions in their code are vulnerable.



Malicious Code

Figure 4.4: Result for malicious code.

In-App Examples

As an extra feature that can be used by not-so-advanced users to try out PHP or by an advanced user to benchmark on all dialects, we add a drop-down list to the navigation bar. This list contains famous algorithms and problems that are encountered in programming.

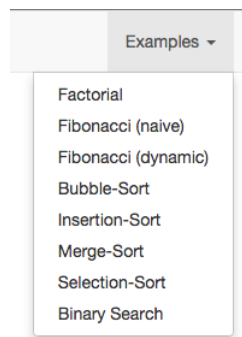


Figure 4.5: Examples drop-down list.

Pressing any one of the items on the drop-down list will replace the code in the editor with the code received from the server for that item. We can add new items to the list at any time.

```
1- function bubblesort($data) {
2   $data_length = count($data);
3-   for ($i = 0; $i < $data_length; $i++) {
4-     for ($j = 0; $j < $data_length - 1 - $i; $j++) {
5-       if ($data[$j + 1] < $data[$j]) {
6-         $data = swappositions($data, $j, $j + 1);
7-       }
8-     }
9   }
10  return $data;
11 }
12
13- function swappositions($data, $left, $right) {
14   $backup_old_data_right_value = $data[$right];
15   $data[$right] = $data[$left];
16   $data[$left] = $backup_old_data_right_value;
17   return $data;
```

Figure 4.6: The editor after pressing the Bubble-Sort item.

4.1.2 Test Suites

There are two test suites available to the user to run. They are the performance test suite and the behavioural test suite. The tests can be added and removed independently of each other. To run the tests, the user has to navigate away from the main page of the website. The new page has no editor and only displays the test source code and results. The user is not allowed to change the source code in any way. This allows us to run the tests without any special restrictions since we wrote the test suites.

4.2 Server-Side

The server-side must handle all of the requests made by the client-side and carry out the necessary security operations on the PHP code. The server-side of our application is written in PHP (Zend). Writing our server-side in PHP would give us a better understanding of the language and allow us to develop a more secure environment to execute user's PHP code.

4.2.1 Web server

Our web application is currently running on a single virtual machine on the departmental "DoC Private Cloud" on the Cloudstack servers. The server runs Ubuntu 14.04 LTS. We chose to use Ubuntu because we have lots of experience with it already (especially running it as a web server) and because its official repositories contain all of the packages we need. For our website, we need to install a various range of libraries and packages so that the four interpreters may work, and Ubuntu 14.04 is compatible with all the required dependencies.

Our web application makes use of the Apache HTTP Server. Apache works well with PHP and is easy to setup. The web server itself is run as *root*, but the worker processes (those that handle HTTP requests) run as the non-privileged user *www-data*. The username which runs the worker processes is extremely important to our web application because we will use it in the jail when executing the user given PHP scripts and to set strict read/write permissions.

4.2.2 chroot Jail

The design of the jail does not concern the user but is crucial to running the user's PHP scripts securely without harming our server. We use a specific type of chroot known as *schroot* (securely change root). *schroot* allows a user to enter a chroot jail without requiring root credentials, automatically mounts important file systems inside the chroot, and allows the user to create sessions from a chroot that function like an independent chroot that disappear when the user is finished, leaving the source chroot in its original condition.

Before setting up the web server, we add a folder with four subfolders as shown in blue in figure 4.7 to the jail.

The read/write permissions inside the jail are extremely strict. The *www-data* user only has read/write permissions to the newly created folder *tmp* and its children. Another file that the *www-data* user has unrestricted access to is *.hhvm.hhbc*. *hhvm.hhbc* is a file which holds the intermediary byte code for HHVM. *www-data* will need access to this if we are to run HHVM (also Hack, since Hack is run through HHVM).

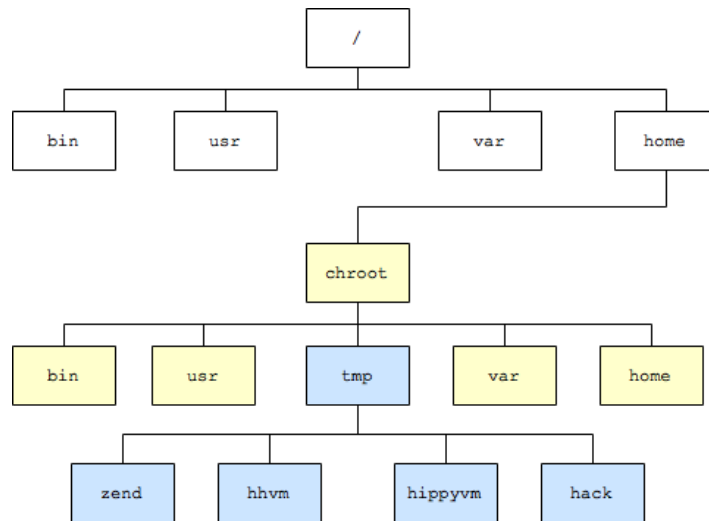


Figure 4.7: Modified filesystem of the chroot jail.

A challenge with the `.hhvm.hhbc` file is that it is created at runtime when the HHVM compiler is invoked and `www-data` will not have read/write to a newly created file due to the strict user-permissions. One way to overcome this is to create an empty file with the name `.hhvm.hhbc`. This will momentarily fix the problem but once the current session is closed, all files that were created in the session will be deleted including the `.hhvm.hhbc` file. This can be solved by creating the `.hhvm.hhbc` file whenever a new session is started.

An argument can be made that if we just create the `.hhvm.hhbc` in the source chroot environment as the root so that it exists before a session is started and will remain after the session has ended. This will not work as the `.hhvm.hhbc` needs to be created inside the `~user/` directory and the user for chroot and schroot differ. Only the root user has permission to create a file inside a chroot jail (note that this is different to schroot) so if we were to create the `.hhvm.hhbc` file as the root user, it would be created in the incorrect `~user/` directory since the root user has access to the entire filesystem including that out of the jail.

4.3 Architecture

In this section we will explain the architecture of the web application. This section will mainly show how the entire application works, starting from user input to how the input is handled and a result is sent to the client. Most of what our application's operations take place on the server abstracted from the user. This is why we have divided our server into four parts, the Process Router, the Jail, the Input Handler and the Output Handler. With such an approach the entire backend is not only easier to manage and debug but also creates a nice assembly line model.

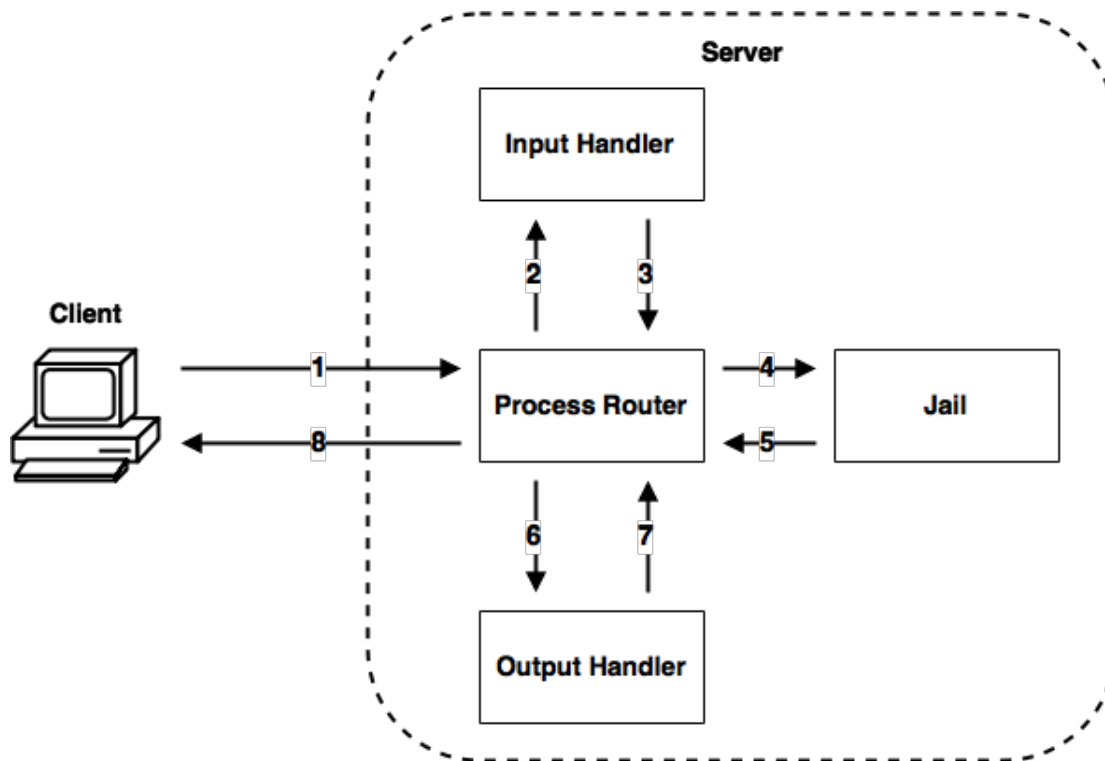


Figure 4.8: System flow while executing valid PHP code.

Figure 4.8 shows the entire process that takes place within our web application. Each step is described in greater detail below:

- 1 The client-side sends all the relevant information to the server-side. The information includes the user's snippet of PHP code and boolean

values for which interpreters are selected.

- 2 The Process Router sends the user's PHP code to the Input Handler.
- 3 After the Input Handler receives the PHP code, the first thing it does is scan the code for functions that are blocked in our application. If any such functions are found, a message is sent to the Process Handler informing it as well. If the PHP code passes through the scanner, we add util variables and functions to the start and end of the code so that we can extract relevant information (execution time) and impose restrictions. At this point, the modified PHP code is sent back to the Process Router.
- 4–5 Depending on the response from the Input Handler, the Process Router will either return negative results to the client-side or carry on with the regular process. Depending on which interpreters the user has chosen, the Process Router will save the modified PHP code to the appropriate subfolders in the tmp folder inside the jail. The code is saved in a file using the client's IP address. This allows for multiple users to use the web application without any problems. After the code has been saved, the chosen interpreters will be run inside the chroot jail (using schroot for *www-data*). The outputs and exit codes from the executions are returned to the Process Router from the jail. Finally, the file that was created earlier is deleted.
- 6 The Process Router checks the exit code and decides whether to extract the time it took to execute from the information returned from the jail. If the exit code is 0, meaning the script executed successfully, the variable that holds the time for that dialect is set. Afterwards, the output is sent to the Output Handler.
- 7 The Output Handler scans the output for information disclosure threats. It goes through the code and removes all text associated with our system and sends the modified output to the Process Router.
- 8 At this point, the code has been securely executed and the output has been scanned and modified as necessary. The output and the time taken for the execution is sent back to the client-side in a JSON² format.

²<http://json.org/>

Chapter 5

Implementation

In this chapter we discuss the implementations of the components that make up our client-side and server-side. We explain how we set up the chroot jail so that it worked with all four dialects. We also describe in detail the Input and Output Handlers that perform operations on the user's PHP code.

5.1 Client-Side

The client-side of our web application was always focused on user interface and design which has already been explained in chapter 4. Most of the implementation was trivial involving performing asynchronous HTTP (Ajax) requests with the server and writing HTML and CSS code to set up the front-end. The only complex operation that takes place on the client-side is an extra feature we added to achieve a more positive user experience that other websites were missing.

The feature is that we allow the user to choose whether to start their script with the `<?php` tag or not. On other websites, first-time users will often incorrectly start their scripts because those websites have no indication whether a `<?php` tag is required or not. This combined with the issue that websites navigate away from the editor to a static page for results is an unpleasant experience for the user as the user has to navigate back to the editor and retype their code. This experience is completely avoided on our website. The

only overhead of this feature is that we need to send an extra boolean value to the server so that the Output Handler can use it to show the correct line numbers when showing errors.

5.2 Setting up Our chroot Jail

The chroot jail is one of the main factors in keeping our server secure and therefore it has to be set up with the utmost care. The design of the file structure has already been explained in subsection 4.2.2. Here we will describe the process in setting up our custom jail with all four dialects.

To set up our jail, we (as root user) set up a folder on our server to hold our chroot jail. We used the `debootstrap` command to create a full but minimal Debian installation inside the chroot jail. The Linux distribution we have set up inside the jail is Ubuntu 14.04 with amd64 architecture.

```
1 [jail]
2 description=ubuntu_jail
3 type=directory
4 directory=/srv/chroot/jail
5 users=sak212,www-data
6 groups=gernot
7 root-users=root
```

Listing 5.1: Our configuration file for the jail.

After setting up the a minimal distribution in the jail, we installed the four dialects. It is important to chroot into the jail as the root and install the required dependencies and packages so that they persist in the jail when a non-root user enter the jail via `schroot`.

Installing the native PHP compiler (Zend version 5.5.9) was relatively easy and all the dependencies get installed automatically.

HippyVM required a full source checkout of RPython into the chroot jail which was undesirable as we want to restrict the resources in the jail. However, the HippyVM builds on the system and produces a `hippy-c` binary that

works mostly like a `php-cli`¹. Therefore, when the build process is complete, we can restrict read/write permissions to RPython and only use the `hippy-c` binary for execution.

Installing HHVM (version 3.7.1) in the chroot jail was a challenge. The compiler had an enormous number of dependencies which did not install on the chroot jail automatically and therefore had to be installed manually. Even then, various libraries had problems linking. Once again, the job had to be done manually.

Hack did not require any installation as it runs on HHVM but it does need a configuration file to start the type checker. We create an empty file named `.hhconfig` in the folder where the type checker is invoked. The type checker uses this file to know what directory tree to type check without needing to specify paths to every command. When running the Hack type checker, it looks in the current directory for the `.hhconfig` file. If the `.hhconfig` file is not found in the current directory, the directory will be traversed upward looking for that file until it finds one.

5.3 Process Router

The Process Router's has two main responsibilities. It controls the pipeline for the entire system and executes the user's PHP code by entering the jail as `www-data` and invoking the compilers on the command line. It sits at the centre of the entire application and communicates with all components of the application. The routing mechanism behaves as described earlier.

The execution of the user's PHP code is done through the `exec()` function in PHP. The function will return an array containing every line from the output of executing the user's PHP code. The function also returns the exit code from the execution which is used to set the time variables. The time variables hold information on how long the execution took and are only set if the code executed successfully.

After the entire process is complete, the Process Router encodes the information that is to be sent to the client-side into JSON. The `is_mali` boolean

¹<http://php.net/manual/en/features.commandline.php>

in listing 5.2 corresponds to the value returned from the Input Handler when scanning for malicious code.

```
1 {
2   "is_mali":false,
3   "zend_out":"Hello, World!<br>",
4   "zend_time":"0.00002799s",
5   "hhvm_out":"Hello, World!<br>",
6   "hhvm_time":"0.00057109s",
7   "hippyvm_out":"Hello, World!<br>",
8   "hippyvm_time":"0.00002109s",
9   "hack_out":"Hello, World!<br>",
10  "hack_time":"0.00057209s"
11 }
```

Listing 5.2: JSON returned to client side.

5.4 Input Handler

As mentioned before, the Input Handler serves two functions. It acts as a scanner that detects malicious code and modifies the code to make it ready for execution. These two steps are described in detail below.

5.4.1 Scanning Phase

The scanning phase is vital to our security. It detects and stops harmful code from reaching our server.

The scanning process is broken down into smaller steps. First, the user's PHP code is converted into tokens using PHP's very own `token_get_all()`. The tokens are then parsed and the tokens which hold relevant information like the names of functions are retrieved. These tokens include `T_STRING`, `T_ENCAPSED_AND_WHITESPACE`, `T_EVAL`, etc. These tokens will contain their id which corresponds to the token itself and a string that equals the name of the function or the name of the variable. To have a better understanding of the tokens see Appendix.

The filtered tokens are now scanned for exploitable functions. This list of exploitable functions is predefined on our server. The list comprises of a large number of functions that we do not want the user to have access to. The list is described in greater detail later in the section.

```
1 $predefined_list; \\ list containing blocked funtions
2 $tokens = $token_get_all($source);
3
4 $banned_functions_found;
5
6 foreach ($token in $tokens) {
7     if ($token is NOT empty AND holds relevant information) {
8         foreach ($func in $predefined_list) {
9             if ($token.information is equal to $func) {
10                 $banned_functions_found.add($func);
11                 break;
12             }
13         }
14     }
15 }
```

Listing 5.3: Pseudocode for scanning.

Following the Pseudocode shown in Listing 5.3, we produce a list of blocked functions that we found in the user’s PHP code. If the list is empty, we move onto the next phase, but if it is not the entire process stops and the list of blocked functions is returned to the client-side where it is shown to the user so that they may change their code and try again.

Blacklisted Functions

The list was put together after consulting research papers and the publicly available source code of a security software² on GitHub. The main functions that are blocked are described below.

- **Command Execution** – We do not want the user to have access to the shell as they might try to tamper with the system or try to break out of the jail.

²<https://github.com/robocoder/rips-scanner>

- **PHP Code Execution** – Apart from eval there are other ways to execute PHP code: include/require can be used for remote code execution in the form of Local File Include³ and Remote File Include⁴ vulnerabilities.
- **Functions which Accept Callbacks** – These functions accept a string parameter which could be used to call a function of the attacker’s choice. Depending on the function the attacker may or may not have the ability to pass a parameter. In that case an Information Disclosure function like phpinfo() could be used.
- **Information Disclosure** – Most of these function calls are not sinks. But rather it maybe a vulnerability if any of the data returned is viewable to an attacker. If an attacker can see phpinfo() it is definitely a vulnerability.
- **Filesystem Functions** – Our jail will not allow the user to have any affect on the filesystem but these functions are still blocked as a precaution.

Variable Function Attacks

PHP supports the concept of variable functions. This means that if a variable name has parentheses appended to it, PHP will look for a function with the same name as whatever the variable evaluates to, and will attempt to execute it. Fortunately for us, variable functions won’t work with language constructs such as echo, print, eval(), unset(), isset(), include, require and the like. However a wrapper function can be utilised to make use of any of these constructs as variable functions.

If an attacker does use a wrapper function to use a blocked function, like eval() for example, our scanner will find it when searching through the tokens. To get around the scanner, an attacker may try to deceive the scanner by assigning the variable to a blocked function or a wrapper at runtime like in listing 5.4, where the function eval() is broken into different strings and concatenated before being assigned to a variable. This will not work in PHP, as the interpreter will try to look for a user defined function called eval.

³<https://www.exploit-db.com/exploits/12510/>

⁴https://en.wikipedia.org/wiki/File_inclusion_vulnerability

```
1 <?php
2     function foo($bar) {
3         $var = "ev" . "al";
4         $var($bar);
5     }
6
7     foo("malicious code");
8 ?>
```

Listing 5.4: PHP code making use of variable functions.

5.4.2 Modifying Phase

The modifying phase is a relatively small and quick but an extremely important one in the process. During this phase, we modify the user's PHP code by adding our own PHP code to the start and end of the code.

The main modifications we make to the user's PHP code are:

- **Open and close tags** – We add the opening and closing tags to the user's PHP code. Hack requires `<?hh` as an opening tag.
- **Time keeping variables** – These variables are used to keep track of the execution time for the script so that we can show it to the user on the client-side later.
- **set_time_limit()** – This function is used by PHP to limit the maximum execution time. By default the limit is 30 seconds which is too high for our server. We do not want users abusing our server's resources so we set the maximum execution time to 3 seconds.
- **ini_set()** – We use this function to set a memory limit on the current PHP script. Once again, we do not want users to take advantage of our system's resources so we set it to 64K.

Listing 5.5 shows the modifications made to the file after it leaves the modifying phase. The `printf` statement is used to output the time taken onto the shell so that it can be retrieved.

```
1 <?php
2     $time_before_$ip = microtime(true);
3     set_time_limit(3);
4     ini_set('memory_limit','64K');
5
6     /* User Code */
7
8     $time_after_$ip = microtime(true);
9     printf('%.7f', $time_after_$ip - $time_before_$ip);
10 ?>
```

Listing 5.5: The modifications made to the user's code.

HippyVM's Special Case

The code being executed in HippyVM requires an extra modification. HippyVM produces the same exit code on failure and success. Therefore, we need to output a unique value to act as the exit code for HippyVM. Printing an extra value at the end of execution shows that the end of the file was reached successfully. This allows the Process Handler to carry out its operations.

The Namespace Case

When a namespace declaration statement is present, it needs to be the very first statement in the script. However, with our modifications will cause scripts with namespaces to fail. Whenever we encounter this case, we simply start modifying the user's PHP code after the namespace declaration.

This special case is one of the examples where the knowledge we obtained from using PHP as our backend scripting language allowed us to create a better web application.

5.5 Output Handler

The Output Handler works much like the scanner phase in the Input Handler. It scans the output for any information relating to our file system, system architecture or configurations. Revealing such information makes us vulnerable to information disclosure threats, therefore, we remove it from the final output.

Due of our modifications to the code earlier, the line numbers that are returned are incorrect. The Output Handler replaces the incorrect line numbers with the correct ones. It does so by calculating the offset created by our modifications and whether the user used a `<?php` tag to start their php code. The offset is added to the line number returned from execution and that gives us the correct line number.

Figure 5.1 shows the output when the Output Handler is not active. It can be seen that information about the directories is revealed to the user. However, when the Output Handler is active in figure 5.2, no such information is revealed and the line numbers have also changed.

```
Zend: PHP Parse error: syntax error, unexpected '$output' (T_VARIABLE) in  
/home/tmp/zend/129_31_188_9code.php on line 16
```

```
HHVM: Fatal error: syntax error, unexpected T_VARIABLE in /home/tmp/hhvm/129_31_188_9code.php on line 16
```

Figure 5.1: Output without using the Output Handler.

```
Zend: PHP Parse error: syntax error, unexpected '$output' (T_VARIABLE) on line 12
```

```
HHVM: Fatal error: syntax error, unexpected T_VARIABLE on line 12
```

Figure 5.2: Output with using the Output Handler.

Chapter 6

Tests

In this chapter we discuss the tests that we have written for the dialects and go into further detail for some of the tests. The main purpose of the tests is to compare how the four dialects perform against each other and to showcase how their behaviour differs to one another. Note that HHVM and HippyVM are still being developed so we will not be testing features that are not compatible with all the dialects (HHVM is almost fully compatible with the PHP specification but HippyVM is not).

These tests are not run inside a jail and do not have any restrictions placed on them. We are writing the tests ourselves so we can assume that they are not malicious. HHVM and Hack will be combined for all tests as they are identical in behaviour and performance.

6.1 Performance Tests

The performance tests we run for benchmarking are based on PHP's internal benchmarking tests. We run each test five times, discard the two slowest runs and use the remaining three times to get an average time. This way we remove the overhead of the JIT compilers and are able to get more accurate results. We now go over some of the performance tests we used. All execution times shown below are normalised to the fastest time so that we can see the performance difference between the dialects more clearly.

Function Calls

This is a very simple test where a user defined function is called a large amount of times. The user defined function does not do anything.

```

1 function test($arg) {}
2
3 function call_to_usr_func() {
4     for ($i = 0; $i < 1000000; ++$i)
5         test("hello");
6 }

```

Zend	HHVM/Hack	HippyVM
26.477001	2.480766	1

Figure 6.1: Test using function calls shows HippyVM to be the fastest.

String Operations

This test measures the performance for concatenation operations on a string.

```

1 function string_concat($n) { // 200000
2     $str = "";
3     while ($n-- > 0) {
4         $str .= "test\n";
5     }
6     $len = strlen($str);
7 }

```

Zend	HHVM/Hack	HippyVM
2.450905	1.82025	1

Figure 6.2: Test using string concatenations shows HippyVM to be the fastest.

Array Operations

In this test, we are measuring the performance of reading from and writing to an array.

```

1 function array_op($n) { // 50000
2   for ($i = 0; $i < $n; ++$i) {
3     $X[$i] = $i;
4   }
5   for ($i = $n - 1; $i >= 0; --$i) {
6     $Y[$i] = $X[$i];
7   }
8 }

```

Zend	HHVM/Hack	HippyVM
3.580284	1	3.004363

Figure 6.3: HHVM is fastest when operating on arrays.

Recursion

To test recursion, we use the naive way to find fibonacci numbers.

```

1 function fib_rec($n) {
2   return(($n < 2) ? 1 : fib_rec($n - 2) + fib_rec($n - 1));
3 }
4
5 function fib($n) { // 30
6   $res = fib_rec($n);
7 }

```

Zend	HHVM/Hack	HippyVM
17.707875	1	9.771705

Figure 6.4: HHVM recurses the fastest.

Sorting

To test sorting, we use heap-sort. A randomly generated array of length 20000 is passed to the function to sort.

```

1 function heapsort($n, &$ra) { // 20000, randomly array
2     $l = ($n >> 1) + 1;  $ir = $n;
3     while (1) {
4         if ($l > 1) {
5             $rra = $ra[--$l];
6         } else {
7             $rra = $ra[$ir];
8             $ra[$ir] = $ra[1];
9             if (--$ir == 1) {
10                $ra[1] = $rra;
11                return;
12            }
13        }
14        $i = $l; $j = $l << 1;
15        while ($j <= $ir) {
16            if (($j < $ir) && ($ra[$j] < $ra[$j + 1])) {
17                $j++;
18            }
19            if ($rra < $ra[$j]) {
20                $ra[$i] = $ra[$j];
21                $j += ($i = $j);
22            } else {
23                $j = $ir + 1;
24            }
25        }
26        $ra[$i] = $rra;
27    }
28 }

```

Zend	HHVM/Hack	HippyVM
2.147931	1	2.747962

Figure 6.5: HHVM is the fastest at sorting arrays.

Mandelbrot set

We use the Mandelbrot set as a test for the dialects. It is a complex object in mathematics and requires a large number of operations so it makes for a good test.

```

1 function mandel() {
2     $w1 = 50; $w2 = 40; $h1 = 150; $h2 = 12;
3     $recen = -.45; $imcen = 0.0;
4     $r = 0.7; $s = 0; $x = 0; $y = 0;
5     $rec = 0; $imc = 0; $re = 0; $im = 0; $re2 = 0; $im2 = 0;
6     $color = 0;
7     $s = 2 * $r / $w1;
8     for ($y = 0 ; $y <= $w1; $y++) {
9         $imc = $s * ($y - $h2) + $imcen;
10        for ($x = 0 ; $x <= $h1; $x++) {
11            $rec = $s * ($x - $w2) + $recen;
12            $re = $rec;
13            $im = $imc;
14            $color = 1000;
15            $re2 = $re * $re;
16            $im2 = $im * $im;
17            while((((($re2 + $im2) < 1000000) && $color > 0)) {
18                $im = $re * $im * 2 + $imc;
19                $re = $re2 - $im2 + $rec;
20                $re2 = $re * $re;
21                $im2 = $im * $im;
22                $color = $color - 1;
23            }
24        }
25    }
26 }

```

Zend	HHVM/Hack	HippyVM
13.842321	1.225445	1

Figure 6.6: HippyVM performs the Mandelbrot test the fastest.

6.2 Behavioural Tests

The behavioural tests are far more interesting than the performance ones. It is exciting to see how the dialects differ and what is going on under the hood that makes them behave differently. We will look at a few tests that produce different behaviour from the interpreters.

These sort of tests are challenging to write as the developers of the dialects are constantly trying to be as compatible with the PHP language specification as possible and that makes finding behaviour that is actually different difficult. We wrote our tests after consulting various resources, which included the research paper *An executable formal semantics of PHP* by Daniele Filaretti and Sergio Maffei. We also looked at a large number of tests published on GitHub^{1,2} by the development teams behind the interpreters.

For the behavioural tests we will change the way we list our tests so that we can interact better with the source code in this section.

Aliasing

Consider the following test code from the research paper mentioned earlier:

```
1 $a = array("a", "b", "c");
2
3 foreach ($a as &$v) {}; // aliasing on $v
4 foreach ($a as $v) {};
```

Calling `var_dump()` on `$a` produces:

¹<https://github.com/php/php-src/tree/master/Zend/tests>

²<https://github.com/facebook/hhvm/tree/master/hphp/test>

```

// Zend, HHVM/Hack
var_dump($a);
> array(3) {
  [0]=>
  string(1) "a"
  [1]=>
  string(1) "b"
  [2]=>
  &string(1) "b"
}

// HippyVM
var_dump($a);
> array(3) {
  [0]=>
  string(1) "a"
  [1]=>
  string(1) "b"
  [2]=>
  string(1) "b"
}

```

The output may seem strange as the values in the array are changing even though there is no code inside the bodies of the two loops. What happens is that variable `$v` from the first loop has global visibility and when the first loop ends `$v` and `$a[2]` are aliased. Now at every iteration in the second loop, the assignment `$v = $a[...]` is made, storing the current array element in `$v` and hence in `$a[2]`. Because of this we know that the last element in the array should be aliased.

When we look at the output, Zend and HHVM/Hack behave similarly but HippyVM behaves differently. We can see that third element in the array for Zend and HHVM/Hack is aliased like it should be while it is not for HippyVM. If we add the line `$v = "x"`; to the end of the code and run it. We should expect the last element in the array to change to "x". Lets see:

```

// Zend, HHVM/Hack
var_dump($a);
> array(3) {
  [0]=>
  string(1) "a"
  [1]=>
  string(1) "b"
  [2]=>
  &string(1) "x"
}

// HippyVM
var_dump($a);
> array(3) {
  [0]=>
  string(1) "a"
  [1]=>
  string(1) "b"
  [2]=>
  string(1) "x"
}

```

The last element changes as expected due to the aliasing. Even though it does not show on the HippyVM output, it is still aliased as the value changed. Since `$v` and `$a[2]` are aliased, we should expect the value of `$v` to change

if we modify `$a[2]`. Now instead of `$v = "x"`, lets try `$a[2] = "x"` and call `var_dump` on `$v`. We should expect to see the same result as before:

```
// Zend, HHVM/Hack          // HippyVM
var_dump($v);              var_dump($v);
> string(1) "x"            > string(1) "b"
```

The value of `$v` in Zend and HHVM/Hack changed as expected but stayed the same in HippyVM. This means that in HippyVM aliasing is one-sided while it is two-sided in Zend and HHVM/Hack.

Another test that shows the same aliasing behaviour for HippyVM is:

```
1 $array1 = array(1,2);
2 $x = &$amp;array1[1]; // Unused reference
3 $array2 = $array1; // Aliased to $array2!
4 // unset($x);
5 $array2[1]=22;
```

Calling `var_dump($array1)` produces:

```
// Zend, HHVM/Hack, HippyVM
var_dump($array1);
> array(2) {
    [0]=>
    int(1)
    [1]=>
    &int(22)
}
```

The output is the same for all the dialects. Notice the aliasing on the second element of the array. `$x` may be an unused reference but when we copy `array1` into `array2`, the alias applies to `array2` as well. So when we set `$array[2] = 22`, the value of `$x` is also 22.

Now if we uncomment the `unset($x)` function call in the test and run it, we get:

```
// Zend, HHVM/Hack                // HippyVM
var_dump($array1);                var_dump($array1);
> array(2) {                       > array(2) {
  [0]=>                             [0]=>
  int(1)                             int(1)
  [1]=>                             [1]=>
  int(2)                             &int(22)
}
```

The second element in the array for Zend and HHVM/Hack is 2 (not aliased) which the second element in the array HippyVM is 22 (aliased). Unsetting `$x` stops the aliasing between `$x`, `$array1[1]` and `$array2[2]` in Zend and HHVM/Hack but not HippyVM. Calling `unset` on `$x` destroys the variable in HippyVM as well but to HippyVM the alias was called on `$array1[1]`. Therefore, even if `$x` is unset, `$array1[1]` and `$array2[1]` are still aliased. To stop aliasing, we would need to unset `$array1[1]`.

In the first test for aliasing, we saw a variable not in the array aliased on and in the second test we saw a element in the array aliased on.

Internal Array Pointers

In PHP, arrays have internal pointers that are used for traversal by loops, iterators, etc. Arrays have functions that allow users to change the pointers as they wish.

This is a test that makes use of the internal pointer of an array:

```
1 $arrayOuter = array("key1", "key2");
2 $arrayInner = array("0", "1");
3
4 while(list(, $o) = each($arrayOuter)) {
5     // $placeholder = $arrayInner;
6
7     while(list(, $i) = each($arrayInner)) {
8         echo "$o, $i\n";
9     }
10 }
```

Running the test produces:

```
// Zend, HHVM/Hack, HippyVM

> key1, 0
key1, 1
```

All the dialects produce the same output but this behaviour is still strange. The code never enters the second while loop while in the second loop of the first while loop. To investigate, we call `var_dump(current($arrayInner))` which returns `bool(false)` during the second iteration of the first while loop. The `current()` function simply returns the value of the array element that is currently being pointed to by the internal pointer.

This strange behaviour happens because after the second while loop is done iterating, the internal pointers of `$arrayInner` do not reset to the start. Therefore, when the second iteration of the first while loop begins, the current pointer has already traversed the entire array and points beyond the end of the array causing it to return as false.

This behaviour may be strange but is common to all the dialects. It gets interesting when we uncomment line 5 from the test case and run it:

```

// Zend
> key1, 0
key1, 1
key2, 0
key2, 1

// HHVM/Hack, HippyVM
> key1, 0
key1, 1

```

Making a copy of the inner array before it is traversed causes the internal pointer of the array to point to the start of the array. This behaviour only exists in Zend and could be a decision made by the developers. The array resets its own internal pointer before being copied so that the copied array's internal pointer is also pointing to the start. Perhaps the developers of HHVM/Hack and HippyVM believed that the copied array should be exactly like the old one and not reset its internal pointer.

There is a simple test associated with internal pointers of arrays that shows what happens to the pointers after the array is fully traversed.

```

1 $arr = array('one', 'two', 'three');
2
3 foreach($arr as $key => $value);
4 $curr = current($arr);
5
6 var_dump($curr);

// Zend
> bool(false)

// HHVM/Hack
> string(3) "one"

```

HippyVM has a segmentation fault during this test. This is probably because the internal pointer is pointing to somewhere in memory we do not have access to. In other words, the internal pointer is pointing beyond the end of the array just like Zend. HHVM/Hack, however, resets its internal pointer to the start after it reaches the end of the array.

Foreach Loop

To get a better understanding of the foreach loop, we use it in conjunction with the current function in the following test:

```
1 $arr = array("A", "B", "C", "D");
2
3 foreach ($arr as $v) {
4     var_dump(current($arr));
5 }

// Zend                                     // HHVM/Hack, HippyVM
> string(1) "B"                               > string(1) "A"
string(1) "B"                               string(1) "A"
string(1) "B"                               string(1) "A"
string(1) "B"                               string(1) "A"
```

The current function is a by-ref function, even though it does not modify the array. It has to be in order to work well with all the other functions like next which are all by-ref. The reason that the "B" and "A" are repeating is that the array that is being iterated over by the foreach is different to the one that current is being called on. The foreach loop makes a copy of the array and iterates over that therefore the internal pointer is not moving. In Zend, the reason why we get a "B" instead of an "A" is that foreach advances the array pointer before running the user code, not after. So even though the code is at the first element, foreach already advanced the pointer to the second. The foreach implemented in HHVM/Hack and HippyVM seems to be advancing the array pointer after running the user code like a more conventional language.

PHP allows you to substitute the iterated entity during the loop. So you can start iterating on one array and then replace it with another array halfway through. Or start iterating on an array and then replace it midway with an object:

```
1 $arr = [1, 2, 3, 4, 5];
2 $obj = (object) [6, 7, 8, 9, 10];
3
4 $ref = &$arr;
5 foreach ($ref as $val) {
6     echo "$val\n";
7     if ($val == 3) {
8         $ref = $obj;
9     }
10 }

// Zend                                // HHVM/Hack, HippyVM
> 1 2 3 6 7 8 9 10                       > 1 2 3 4 5
```

The Zend interpreter behaves as expected. It just starts iterating the other entity from the start once the substitution has happened. HHVM/Hack and HippyVM never switch to the object and continue with the array till the end.

If we call `var_dump` on `$ref` after it has been substituted we get:

```
> object(stdClass)#1 (5) {
  [0]=>
  int(6)
  [1]=>
  int(7)
  [2]=>
  int(8)
  [3]=>
  int(9)
  [4]=>
  int(10)
}
```

The value of `$ref` does switch to the object but the entity that the `foreach` is iterating over does not change. The `foreach` in HHVM/Hack and HippyVM makes a copy of the entity it is iterating over and continues to iterate over it even if the original is replaced.

Evaluation Order

To check the evaluation order of the dialects, we use a test from the Zend test suite:

```
1 function i1() {
2     echo "i1\n";
3     return 1;
4 }
5 function i2() {
6     echo "i2\n";
7     return 1;
8 }
9 function i3() {
10    echo "i3\n";
11    return 3;
12 }
13 function i4() {
14    global $a;
15    $a = array(10, 11, 12, 13, 14);
16    echo "i4\n";
17    return 4;
18 }
19
20 $a = 0;
21 list($a[i1()+i2()], , list($a[i3()], $a[i4()]), $a[]) =
    array (0, 1, array(30, 40), 3, 4);
22
23 var_dump($a);
```

The output from the calls to `echo` will show us in what order the evaluation during assignments:

```

// Zend, HHVM/Hack
>i1 i2 i3 i4
array(6) {
  [0]=>
  int(10)
  [1]=>
  int(11)
  [2]=>
  int(0)
  [3]=>
  int(30)
  [4]=>
  int(40)
  [5]=>
  int(3)
}

// HippyVM
> i4 i3 i1 i2
array(5) {
  [0]=>
  int(10)
  [1]=>
  int(11)
  [2]=>
  int(0)
  [3]=>
  int(30)
  [4]=>
  int(40)
}

```

It can be seen from the output that Zend and HHVM/Hack follow the same order while HippyVM does not. Zend and HHVM/Hack go from left to right for outside the brackets and inside. HippyVM goes right to left for outside but goes left to right inside brackets for expressions.

Namespace

This test shows how the namespace are handled differently in the dialects.

```

1 namespace Foo;
2 define('Foo\\true', false);
3
4 if (true) {
5     echo "TRUE";
6 } else {
7     echo "FALSE";
8 }

```

```

// Zend, HippyVM
> FALSE

// HHVM/Hack
> TRUE

```

Language Constructs

This test shows how differently a function defined in the PHP language specification behaves under different dialects. The function used is `func_get_arg` which gets the specified argument from a user-defined function's argument list. The test case is:

```
1 function foo($a)
2 {
3     $a = 5;
4     return func_get_arg(0);
5 }
6 echo foo(2) . "\n";

// Zend, HippyVM           // HHVM/Hack
> 2                         > 5
```

The `func_get_arg` function, under Zend and HippyVM, returns the value from the arguments without change. HHVM/Hack on the other hand allow the argument being returned to be modified before the `func_get_arg` functions returns it.

Chapter 7

Evaluation

7.1 The Web Application

Our web application securely executes snippets of PHP code in the four dialects that were chosen at the start. The website is up and running on the department's cloud server at the moment. The user interface for our website is clean, minimalistic and easy-to-use. The entire process of executing the code is abstracted from the user. This not only makes it more secure for us but also more convenient for the users as all they have to do is enter their code and press a button; just as if they were on their own machine.

The design of the user interface makes it so that all first time users will know exactly what to do the second they visit the website. The editor we use makes it extremely easy to code in the browser. The editor itself has great reviews online and is used by massive websites like GitHub and Wikipedia. As a way for users to evaluate our website, we have set up a way for the users to send us feedback and bug reports.

Compared to other websites that serve the same purpose as us, we stack up against them quite well. We have our strengths and weaknesses, and so do the other websites. There is no website that runs PHP code in all four dialects but other websites allow their users to save and share code amongst each other, which is a feature that we do not have. Our website remains dynamic the entire time allowing the user to edit their code and run it at

any time without changing pages, while most of the other websites navigate away from the editor and become static while retrieving the result.

7.2 Benchmarks and Test Suites

Our test suites are set up so that test cases can be added and removed at any time without disturbing the rest of the system.

	Zend	HHVM/Hack	HippyVM
Function Calls	26.477001	2.480766	1
String Operations	2.450905	1.82025	1
Array Operations	3.580284	1	3.004363
Recursion	17.707875	1	9.771705
Sorting	2.147931	1	2.747962
Mandelbrot Set	13.842321	1.225445	1

Figure 7.1: Time from running some of our tests.

It can be seen from figure 7.1 that HHVM is the overall fastest by a small margin over HippyVM. Zend performs quite bad compared to the other dialects. Note that HHVM and HippyVM are JIT compilers which have a noticeable delay at startup. HippyVM has shown to have a smaller delay than HHVM. The times we measured in our benchmarks compensated for the delay by discarding the two slowest times.

The behavioural test suite has shown us some strange and interesting characteristics of the interpreters such as the evaluation order that HippyVM follows or how the foreach loop, one the most used commands in PHP, has a different behaviour depending on the dialect.

Overall, we did manage to write test suites but we feel that there should have been more test cases in both the test suites. We underestimated the security challenge of the web application at the start of the project and that cost us time that we could have spent writing more tests. However, the security aspect of the web application was critical to the project.

7.3 Limitations

In this section we discuss the limitations of our application in its current state. We offer potential future improvements and fixes for some of these, but we also believe that some of the limitations cannot be worked around due to the nature of the application.

7.3.1 Scaling Up

Scaling up may prove to be difficult with the current setup. Currently, there is only one virtual machine so there is only one point of attack. If we add more virtual machines, we have to set up chroot jails on all of them and monitor them for attacks. Managing multiple jails is a challenging task and can take up lots of overhead. To scale up, the best solution would be to use Docker¹ instead of a chroot jail.

7.3.2 Security

The PHP language

The PHP language may change at any time and introduce bugs in our Input Handler which scans the user's code for an attack. This can happen at any time and there is no real way to protect against this. All we can do is keep up with the latest PHP specification so our scanner is always up-to-date.

Jails

Jails are a powerful tool, but they are not a security panacea. While it is not possible for a jailed process to break out on its own, there are several ways in which an unprivileged user outside the jail can cooperate with a privileged user inside the jail to obtain elevated privileges in the host environment.

Most of these attacks can be mitigated by ensuring that the jail root is not accessible to unprivileged users in the host environment. To protect against

¹<https://www.docker.com/>

this, we have to make sure that untrusted users with privileged access to a jail should not be given access to the host environment.

Chapter 8

Conclusions

In this chapter we summarise what this project has achieved and where it might go next, providing suggestions for future work and experiments.

Before this project, no web application existed which was capable of securely executing snippets of PHP code on four different dialects but now with our application, we do exactly this. A lot of different components work together to make this application functional. Security was key. There are two lines of security to protect our server. The first is a scanner that looks through the user's code for blacklisted functions and the second is a custom configured chroot jail. All of the user's code is executed inside the jail where it cannot harm the system. Even the output is checked to protect against information disclosure.

We wrote two test suites for the interpreters. One was for benchmarking performance while the other one was to showcase the behavioural differences of the interpreters. The behavioural test suite revealed some quite interesting differences between the dialects.

Our project has provided a much needed, impartial resource for the PHP developer community.

8.1 Future Work

In this section we discuss various improvements that could be made to our project in the future. Many of these are things that we would have liked to implement if more time had been available to work on the project.

- **Saving and sharing code** – We believe that if we want our website to be the widely adopted, this is the main feature that is has to be implemented. On online blog posts and forums, people users regularly share their code through online compilers. With this feature, users will begin to share their code through our application and drive more traffic to our website. In fact, the HHVM team regularly use *3v4l.org* to share their code.
- **Averaging execution times** – In the web application, we should execute the code five times and discard the highest two times returned. This way we remove the overhead of the JIT compilers and are able to get more accurate results for the users.
- **Move to Docker** – To scale up, we should move to Docker. Docker is a tool that can package an application and its dependencies in a virtual container that can run on any Linux server. The main benefit from Docker will be that instead my managing many jails, we would have to manage fewer containers¹.
- **Add more tests** – Adding more test cases would characterise the behaviour of the dialects better.
- **Add more interpreters** – Adding more interpreters, like Recki-CT, will mean we can start writing tests for these interpreters. The benefit of this will be that users can get a hands on with lesser known interpreter.
- **Open source the test suites** – We could open source the test suites on GitHub. Other developers would be able to write their own tests showcasing behavioural differences and make pull requests. After a review, they could be merged into the repository.

¹<https://en.wikipedia.org/wiki/LXC>

Bibliography

- [1] Zend Technologies Inc. An overview on PHP. http://www.zend.com/topics/overview_on_php.pdf.
- [2] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Adrew Paroski, Brett Simmers, Edwin Smith and Owen Yamauchi. The HipHop Virtual Machine.
- [3] Yuko Kashiwagi. PHP. <http://www.edb.utexas.edu/minliu/multimedia/PDFfolder/PHPKashiwagi.pdf>.
- [4] PHP. <http://en.wikipedia.org/wiki/PHP>.
- [5] What is PHP?. <http://php.net/manual/en/getting-started.php>.
- [6] Manual Lemos. Hack Language is All that PHP Should Have Been. <http://www.phpclasses.org/blog/post/230-Hack-Language-is-All-that-PHP-Should-Have-Been.html>.
- [7] PHP Performance. <http://www.phpclasses.org/blog/category/php-performance/>.
- [8] Manual Lemos. PHP compiler performance. <http://www.phpclasses.org/blog/post/117-PHP-compiler-performance.html>.
- [9] W3Techs. PHP Market Report. <http://w3techs.com/technologies/report/pl-php>.

- [10] W3Techs. Usage of server-side programming languages for websites. http://w3techs.com/technologies/overview/programming_language/all.
- [11] Zend API: Hacking the Core of PHP. <http://php.net/manual/en/internals2.zel.zendapi.php>.
- [12] Zend Engine. http://en.wikipedia.org/wiki/Zend_Engine.
- [13] BaroqueSoftware. Hippy. http://www.mlife.pl/pdf/Projekty%20inwestycyjne/Broszura_HIPPY.pdf.
- [14] How the Zend Engine Works: Opcodes and Op Arrays. <http://php.find-info.ru/php/016/ch20lev1sec1.html>.
- [15] Maciej Fijalkowski. Prototype PHP interpreter using the PyPy toolchain - Hippy VM. <http://morepypy.blogspot.co.uk/2012/07/hello-everyone.html>.
- [16] Ori Livneh. How we made editing Wikipedia twice as fast. <http://blog.wikimedia.org/2014/12/29/how-we-made-editing-wikipedia-twice-as-fast/>.
- [17] Allan MacGregor. An Introduction to HHVM. <http://coderoncode.com/2013/07/24/introduction-hhvm.html>.
- [18] Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans and Stepen Tu. The HipHop Compiler for PHP. <http://coderoncode.com/2013/07/24/introduction-hhvm.html>.
- [19] HipHop for PHP. https://en.wikipedia.org/wiki/HipHop_for_PHP#History_Before_HHVM.
- [20] HipHop Virtual Machine. http://en.wikipedia.org/wiki/HipHop_Virtual_Machine.

Appendices

Appendix A

Malicious code tokenised.

```
1 <?php
2
3     $var = "String that decodes into malicious code!"
4
5     eval(gzinflate(str_rot13(base64_decode($var))));
6
7 ?>
```

```
1 Array
2 (
3     [0] => Array
4         (
5             [0] => 372
6             [1] => <?php
7
8             [2] => 1
9         )
10
11     [1] => Array
12         (
13             [0] => 375
14             [1] =>
15
16             [2] => 2
17         )
```

```
18
19 [2] => Array
20   (
21     [0] => 309
22     [1] => $var
23     [2] => 3
24   )
25
26 [3] => Array
27   (
28     [0] => 375
29     [1] =>
30     [2] => 3
31   )
32
33 [4] => =
34 [5] => Array
35   (
36     [0] => 375
37     [1] =>
38     [2] => 3
39   )
40
41 [6] => Array
42   (
43     [0] => 315
44     [1] => "String that decodes into malicious code!"
45     [2] => 3
46   )
47
48 [7] => Array
49   (
50     [0] => 375
51     [1] =>
52
53     [2] => 3
54   )
55
56
57 [8] => Array
```

```
58     (
59         [0] => 260
60         [1] => eval
61         [2] => 5
62     )
63
64 [9] => (
65 [10] => Array
66     (
67         [0] => 307
68         [1] => gzinflate
69         [2] => 5
70     )
71
72 [11] => (
73 [12] => Array
74     (
75         [0] => 307
76         [1] => str_rot13
77         [2] => 5
78     )
79
80 [13] => (
81 [14] => Array
82     (
83         [0] => 307
84         [1] => base64_decode
85         [2] => 5
86     )
87
88 [15] => (
89 [16] => Array
90     (
91         [0] => 309
92         [1] => $var
93         [2] => 5
94     )
95
96 [17] => )
97 [18] => )
```



```
98     [19] => )
99     [20] => )
100    [21] => ;
101    [22] => Array
102        (
103            [0] => 375
104            [1] =>
105
106
107            [2] => 5
108        )
109
110    [23] => Array
111        (
112            [0] => 374
113            [1] => ?>
114
115            [2] => 7
116        )
117
118    [24] => Array
119        (
120            [0] => 311
121            [1] =>
122
123            [2] => 8
124        )
125
126    )
```

	php 5.5	Recki-CT	hhvm 3.2	hippy-c	qb
simple()	139.63357	1.00000	8.30447	7.65693	8.35018
simplecall()	38.99476	FAIL	1.32552	1.00000	FAIL
simpleucall()	54.02041	1.00000	3.52439	1.51072	47.91090
simpleudcall()	52.14534	1.00000	3.75936	1.41614	47.55259
mandel()	21.26249	1.00000	2.03372	2.11208	FAIL
mandel_typed()	23.16553	1.00000	2.11128	2.09212	3.00061
mandel2()	24.43275	1.00000	2.57704	1.87802	FAIL
mandel2_typed()	23.79989	1.00000	2.90105	1.57193	7.11054
ackermann(7)	35.04870	1.00000	2.27557	103.45436	621.72526
ary(50000)	1.39338	FAIL	1.00000	4.47888	FAIL
ary2(50000)	1.26952	FAIL	1.00000	2.28231	FAIL
ary3(2000)	5.96015	FAIL	1.70997	1.00000	FAIL
fibonacci(30)	39.48440	1.00000	1.60647	16.40883	FAIL
hash1(50000)	1.70014	FAIL	1.00000	3.27314	FAIL
hash2(500)	2.23648	FAIL	1.00000	1.30044	FAIL
heapsort(20000)	3.67800	FAIL	1.00000	4.96699	FAIL
matrix(20)	4.38364	FAIL	1.00000	37.72782	FAIL
nestedloop(12)	29.24924	1.00000	2.91459	3.07568	FAIL
sieve(30)	10.95413	FAIL	1.00000	4.95152	FAIL
strcat(200000)	1.48186	FAIL	2.06003	1.00000	FAIL
jumpaluz(50, 50)	11.67746	1.09240	1.48192	1.00000	FAIL
bitpaluz(21)	63.33357	1.00000	21.39655	1.46851	FAIL
bitpaluz2(18)	21.83346	1.00000	6.19715	2.59416	FAIL

Winner, Within Factor Of 2, Within Factor Of 10, > 10 times slower

Figure A.1: Benchmarks of the Recki-CT compiler.