Weighed and Found Legacy: Modernity Signatures for PHP Systems Using Static Analysis

WOUTER VAN DEN BRINK, University of Twente, The Netherlands

The PHP language has undergone many changes in its syntax and grammar, both in what features the language has to offer and in the distribution of language features used by programmers in their projects. We present a novel method of using grammar usage statistics to calculate a modernity signature for a PHP system, so that we can determine its age. The system will aid developers in choosing whether or not to execute or use a PHP system, without having to perform an extensive inspection.

Additional Key Words and Phrases: PHP, static analysis, weighed attribute grammars

1 INTRODUCTION

In its long history and many versions, the PHP language has undergone many changes [5]. One of the first versions of PHP used a Perl-like syntax in HTML comments. The rewrite of the language by Andi Gutmans and Zeev Suraski into an extensible language made it possible for other developers to add new functionality to the language, either by modifying its syntax or by adding new functions and data types.

The language is still evolving nowadays, with the most recent development being the release of PHP 8.1 in November 2021. This version adds many major additions to the syntax, e.g. enumerations [4] and intersection types [1]. These syntax modifications encourage PHP programmers to use new programming paradigms in their code.

Other modifications introduced by new language versions do not modify the syntax, but rather modify the available functions and their signatures. For example, PHP 8.0 introduced the `str_contains()`, `str_starts_with()`, and `str_ends_with()` functions. And there exists a continuing migration from *resource* types to standard class objects, further elaborated by Karunaratne [11].

1.1 PHP Language Levels

For every PHP system, we can define its language level as the minimum major PHP version required to be able to run the code in the system. For example, version 9.11.0 of the Laravel framework requires PHP version 8.0.2 or higher. The language level is then PHP 8.0. Today, information about the minimum required PHP version and other requirements imposed by a PHP system is usually contained in a `composer.json` file, an artifact produced by the Composer package manager[1].

The PHP language level indicated in the `composer.json` file by means of the minimum required PHP version does what it says on the tin: it tells other developers wishing to use a system what version of PHP they should install to run the code. However, it does not tell much about the actual modernity, or rather, the age, of the code base. While PHP regularly has backwards incompatible changes between major versions, much legacy PHP code will still run without problems in later PHP versions, or will do so with few minor modifications.

As a result, it is possible to advertise a code base as being compatible with a recent version of PHP, thereby implying that the system has been recently maintained, while most of the code is in fact very old and might contain several bugs and security issues. The actual modernity of the code is thus invisible to users of the system without performing extensive analysis. Thus, we wish to reliably determine the modernity of a PHP code base without needing to execute the code, and without extensive human inspection.

1.2 Static and Dynamic Analysis

Analyzing software without executing it is called static analysis [3]. Numerous static analysis tools for PHP have been developed and are used today, such as the code inspections offered by PhpStorm [10], PHP AiR [7, 9], and the framework described in [6]. On the other end of the spectrum, dynamic analysis refers to analyzing software while it is being executed [3]. Several dynamic analysis solutions exist for PHP [13, 14]. Note the use of the word *spectrum* here, as most tools, such as the ones cited, in fact employ a hybrid form of static and dynamic analysis to achieve their goals.

Most PHP code analysis tools have a strong focus on standards enforcing and security. The standards enforcing tools notify the user of problems in a code base, like formatting errors or code smells, and some might even propose a solution or apply it autonomously. Tools focused on security either work in a similar way or apply patches or extra logic to code while it is being executed. All these tools have in common that they rely on pattern matching to find points that need attention.

1.3 Research Question

The goal of this paper is to investigate the value of static analysis, particularly grammar usage statistics, in determining the age of a PHP system. More precisely, we will give an answer to the following research question:

To what extent can we use grammar usage statistics to reliably determine the modernity of a PHP system?

We do this by answering the following sub-questions:

RQ1 How can we define a modernity signature of a PHP system using grammar usage statistics?
RQ2 What influences the modernity signature defined in **RQ1**?
RQ3 Can we use the modernity signature of an unknown PHP system to predict its age?

By answering **RQ1**, we create a novel method of applying grammar usage statistics to infer the modernity of a PHP system. The

[1] https://getcomposer.org/

TScIT 37, July 8, 2022, Enschede, The Netherlands

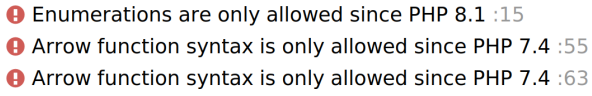


Fig. 1. PhpStorm warns the developer that features are used which are not supported by PHP 7.1.

practice of establishing and analyzing such statistics is further elaborated in Section 3. Then, by answering **RQ2**, we determine to what extent the signature we design is biased, e.g. by the author or the functionality of the code. Finally, with **RQ3** we test the signature in the field by applying it to an unknown PHP system with a known age to be able to discuss the value of the proposed signature.

2 RELATED WORK

Analysis of PHP code bases and the language features being used has been performed before, for example by Hills et al. [8]. In this study, the researchers have performed statistical analysis on feature usage in various open source projects. Various interesting insights and conclusions come forward, but no comments are made on the modernity, or lack thereof, of the code in the corpus.

Current efforts in analyzing PHP systems to determine its age are mostly focused on determining the language level in terms of compatibility. For example, PhpStorm features a static analysis tool to determine whether language features are used which are not supported by the minimum version specified by the developer (see for example Figure 1).

Another example existing solution is PHP Compatinfo [12], a tool which determines the minimum required PHP version and the required installed language extensions for a given PHP system. The tool is quite extensive and has a long history, and it gives us more information on the required PHP version than PhpStorm does. However, it does not give information on the age of the code base it analyses.

3 METHODOLOGY

The modernity signature will use grammar usage statistics to derive the modernity of a code base. Consider, for example, the following grammar definition, which defines the grammar for attributed statements in the PHP language.

```
attributed_statement:
    function_declaration_statement
  | class_declaration_statement
  | trait_declaration_statement
  | interface_declaration_statement
  | enum_declaration_statement;
```

The grammar defines an attributed statement to be the declaration of either a function, a class, a trait, an interface or an enumeration. In other words, there are five possibilities to choose from when creating an `attributed_statement` in this grammar. By analyzing a PHP system, we can add annotations to the grammar describing how often the different paths are taken:

```
attributed_statement:
    function_declaration_statement [33%]
  | class_declaration_statement [48%]
  | trait_declaration_statement [1%]
  | interface_declaration_statement [11%]
  | enum_declaration_statement [7%];
```

In this example, enumerations are chosen in 7% of the cases, but enumerations were introduced in PHP 8.1.0. Thus, we know that this code base will require at least language level 8.1. By adding metadata on the language level associated with the different paths in the grammar, and by adding usage statistics to the different possibilities offered by the grammar, we can infer the modernity signature of a PHP code base.

In the following subsections, we will elaborate the various steps taken to perform the research.

3.1 Gathering a Corpus of PHP Systems

First, various PHP systems have been collected to use in the rest of the research. We have used various open source projects to perform the research. Previous empirical analysis of PHP systems have resulted in many available data sets of open source PHP projects, but they all offer one specific version of the system, as the accompanying studies did not investigate the modernity of systems in general. We could not find existing, freely available data sets of PHP systems, including their historical versions. Thus, for this research, a new corpus had to be assembled, in which we collected historical versions of various open source PHP systems. Specifically, we have gathered historical versions of the open source projects listed in Table 1 and Table 2. The first collection has been used to develop the signature, while the second collection has been used to verify the signature.

Table 1. Open source projects used to train the signature.

Project	Home page	Versions
CakePHP	cakephp.org	29
Joomla!	joomla.org	18
Laravel	laravel.com	16
MediaWiki	mediawiki.org	36
phpMyAdmin	phpmyadmin.net	22
Symfony	symfony.com	32
WordPress	wordpress.org	44

Initially, the goal was to gather, for each project, at least one historical version for every PHP language level for which the authors have released a version. However, it quickly became apparent that the minimum PHP version, and thus the language level, was not published for many historical versions of the various projects. Fortunately, all projects (more or less) adhered to the Semantic Versioning specification [15], and we chose to collect every minor release instead.

3.2 Gathering Grammar Usage Statistics

Next, we have written a library to infer grammar usage statistics of a given PHP system. Previously, PHP AiR[7, 9] has been used to analyze PHP systems. The current version of the underlying parser

Table 2. Open source projects used to test the signature.

Project	Home page	Versions
CodeIgniter	codeigniter.com	8
Guzzle HTTP	guzzlephp.org	37
Monolog	seldaek.github.io/monolog	36
PHPUnit	phpunit.de	57

of PHP AiR[2] only supports PHP 5.2 to PHP 7.2, but we wished to parse and analyze code bases using newer PHP versions as well. The language of the library is PHP, using the PHP parsing library[3] on which the parser for PHP AiR is based.

3.3 Developing and Calculating a Modernity Signature

Having collected grammar usage statistics of the systems in the corpus, we have developed the modernity signature. The signature takes the form of an n-tuple, where n is the amount of PHP language levels the project considers – currently 13[4]. Every element of the tuple is then a quantitative representation of the extent to which the analyzed code base looks like code written for this language level.

The modernity signature was then calculated for every system in the corpus. Based on the results, and the age of the systems in the corpus, the signature was further improved until satisfactory results were achieved. The signature underwent multiple changes during the research. The nature and reasoning of these changes is further elaborated in Section 4.

3.4 Testing the Modernity Signature

Finally, after the modernity signature was developed and calculated for the various systems, we have calculated the modernity signature for projects that were not used in developing the signature, and for which the age is known, so that we can validate the algorithm. For this testing phase, we have used historical versions of the open source projects listed in Table 2. These historical versions were selected and collected using the same method that has been described in Section 3.1.

4 SIGNATURE DESIGN

In this section, we discuss the nature of and reasoning behind the design of the modernity signature.

4.1 Requirements

First, the modernity signature has to be calculated fairly quickly. This is a difficult requirement to satisfy, as parsing is generally a computationally expensive task, and an important component of the signature. The scanner used by the parser library used in this project is PHP's native scanner, which in turn is generated using `re2c` [2, 17]. The performance of this scanner is quite adequate, because it is written in C. Unfortunately, the parser, which transforms the token stream generated by the scanner into an abstract syntax tree (AST), is written in PHP, so we cannot expect impressive parsing speed.

In order to satisfy the performance requirement, it was thus of the utmost importance that the performance quirks and properties of PHP were considered during every step of the development process. An important tool that was used to find and mitigate performance issues in the library is Xdebug[5], along with PhpStorm's debugging tools.

Next, the signature has to produce satisfactory results. The signature takes the form of an n-tuple, with one value for every language level considered by the library, ordered by the age of the level. Every element of the tuple should then represent how much the analyzed code "looks like" code written for that language level. It then follows that the signature should ideally skew more to the left for newer code. Here, skewing to the left means that elements of the tuple representing newer language levels have higher values than others.

Finally, any bias in the signature must be eliminated. The only dependent variable of the signature must be the age of the code base. This is another difficult requirement to satisfy, as naturally, two pieces of source code can differ in their appearance for other reasons than its age. Intuitively, factors like the code's author and intended functionality will influence the distribution of language feature usage, and thus we must ensure that this does not influence the signature significantly.

4.2 Implementation

In this subsection, the eventual method by which the modernity signature is calculated is described. The various choices in the implementation are motivated when needed. The implementation is published as open source software licensed under the MIT license[6].

To determine the modernity signature of a given directory, the analyzer starts by generating a list of files that need to be analyzed. This is done by recursively traversing the directory, and adding any files with `.php` or `.php5` as extension to an array.

Then, for every file in the array, the modernity signature of that file is analyzed. First, the PHP parser is invoked for the file, which generates an abstract syntax tree (AST) of the file. Then, multiple node visitors traverse the tree:

- First, a visitor included in the PHP parser library adds an attribute to every node referencing its parent node. The library does not do this by default, but it is needed for the next visitor.
- Then, a visitor developed for this project determines the minimum and maximum language level for every node in the tree.
- Finally, another visitor calculates the modernity signature of the AST. This is done by generating modernity signatures for every node, then reducing these many signatures to a single signature using weighted sums.

The last two visitors are further explained in the following paragraphs.

[2] https://github.com/cwi-swat/PHP-Parser
[3] https://github.com/nikic/PHP-Parser
[4] The library supports PHP versions 5.2 – 5.6, 7.0 – 7.4, and 8.0 – 8.2.
[5] https://xdebug.org/
[6] https://github.com/WoutervdBrink/PHP-Modernity-Signature

4.2.1 Language Level Determination. The visitor stores the minimum and maximum language level in which the node, in its current form, is supported in attributes of the AST node. Determining these language levels is generally straightforward. In the trivial case, the mere presence of a certain type of AST node is an indicator of the minimum or maximum PHP version. Enumerations were introduced in PHP 8.1, so the minimum and maximum language level of an `Enum` AST node is by definition 8.1 and 8.2, respectively. For some node types, however, determining the language levels of a node depend on the properties of the node. These cases range from simple cases, where new properties were introduced in later versions, to very specific changes in the grammar, like syntactically preventing octal numbers from overflowing[7].

4.2.2 Modernity Signature Reduction. Every node in the AST has zero or more sub nodes. For example, a `PropertyFetch` node, representing the retrieval of the value of a property of a class instance. The node has two sub nodes, namely `var` and `name`. The first sub node represents the class instance for which the property is retrieved, while the other represents the name of the property. We know, from the type declaration of the sub node properties in the parser's source code, that `var` must always be an expression, but that `name` can either be an identifier or an expression.

For every type of AST node, the modernity signature visitor keeps track of how often a certain type of sub node is chosen. For the property fetch example, this would mean that the visitor keeps track how often the source code uses an identifier or an expression as the property name. Furthermore, for every language level, the visitor keeps track how often a sub node is encountered which requires at least this language level to be valid syntax. Thus, the entry for the `PropertyFetch` node in the visitor's memory could be represented as shown in Table 3.

Sub node	Type	Language levels
`var`	Expression (6)	$\langle 0, 2, 3, 1 \rangle$
`name`	Identifier (4)	$\langle 0, 1, 3, 0 \rangle$
	Expression (2)	$\langle 2, 0, 0, 0 \rangle$

Table 3. Example sub node information of the `PropertyFetch` AST node type. Four language levels are considered for brevity.

In an earlier version of the algorithm, we not only incremented the counter for the minimum language level required for the node to be valid syntax, but also for every language level in which the node was still valid syntax. For a node with minimum and maximum language levels of 5.2 and 5.5, this would mean incrementing the counter for language levels 5.2, 5.3, 5.4, and 5.5. This proved to be counterproductive in the training phase, because PHP historically almost never introduces backwards incompatible changes in its grammar. Consequently, in most cases, the counter for *every* language level was incremented, which heavily cluttered the tuple.

For a similar reason, the calculator ignores nodes for which the minimum language version is the oldest language version supported by the library. During development it became clear that most node types have been supported since PHP 5.2. This meant that the value for this language level became very high in comparison to the other values in the signature, which made inspecting the signatures difficult. When the calculator encounters a node with minimum language level 5.2, it acts as if the node is not present in the AST.

After traversing the entire tree, the visitor has information on the distribution of sub node types and their respective language levels, as long as they are used in the source file. Then, the tuples belonging to every sub node type are reduced to a single tuple. This is an iterative process of applying weighted sums of normalized tuples. A normalized tuple is the tuple for which every value is scaled, such that the maximum value is exactly 1. For example, normalizing $\langle 0, 1, 3, 0 \rangle$ results in the tuple $\langle 0, 1/3, 1, 0 \rangle$.

First, the weighted sum is calculated for every sub node. The weight of the tuples is the distribution of its occurrence. In the example shown in Table 3, the weight for a property name as an identifier and as an expression would be $4/6$ and $2/6$, respectively. Thus, the weighted sum for the property name sub node would be calculated as follows:

$$\begin{aligned} &4/6 \cdot \langle 0, 1/3, 1, 0 \rangle \\ +\,&2/6 \cdot \langle 1, 0, 0, 0 \rangle \\ =\,&\langle 1/3, 2/9, 2/3, 0 \rangle \end{aligned}$$

Next, the weighted sum is calculated for every node type. Here, the weight of the tuples is equal to $1/n$, where n is the amount of sub nodes allowed by the AST specification. Continuing with the `PropertyFetch` example, the weights would be equal to $1/2$, as this node has two sub nodes. Furthermore, the tuples are again normalized before calculating the weighted sum.

Finally, the weighted sum is calculated for the entire AST. Now, the weight of the tuples is again equal to the distribution of the occurrence of the node type. If the visitor encountered 74 other nodes while traversing the tree, then the weight of the tuple for the property fetch would be equal to $6/80$. Again, in this weighted sum, the tuples are normalized before the weighted sum is calculated.

So far, we have described how the library calculates the modernity signature for a single file, which is a normalized tuple calculated with a weighted sum of the tuples of the various (sub) nodes. To calculate the signature for an entire directory, i.e. an entire PHP system, again the weighted sum of the normalized tuples is calculated. Now, the weight of the tuple is the ratio of file size to the sum of file sizes. For example, the tuple of a file of 500 bytes in a directory with 5000 bytes of PHP code will be granted a weight of $\frac{500}{5000}$ in the sum.

Note that this method of calculating the weighted sums is mostly based on intuition, rather than for example existing statistical methods or previous research. Fortunately, as will be shown in Section 5, this method shows promising results. Other methods of calculating the weighted sum are left as future work.

5 RESULTS

After the signature calculator was designed and implemented according to the specification described in the previous section, we could let it calculate modernity signatures for the software listed in

[7] https://wiki.php.net/rfc/octal.overload-checking

Table 1 and Table 2. The first list of software was used to design and implement the algorithm. As a rule, we made no further changes to the design or the code when we started calculating the signatures for the second list. In this section, we present our findings. In the next sections, the findings will be further discussed.

As the graphic representations of the results will show, the modernity signatures for two different versions of the same software show little change when their release dates are close together. For this reason, but also simply because a tabular representation of the data would be too large for this report, we have chosen to represent a collection of modernity signatures as a triangular surface plot. Here, the X-axis represents the various language levels, the Y-axis shows the release date, and the Z-axis the values in the modernity signature.

5.1 Training Phase

First, we present the modernity signatures calculated in the training phase, grouped by project in Figure 2. The signatures are all heavily skewed to the right, independent of the release date of the software. This is probably due to the fact that, as explained in Section 4, most node types in the PHP language have been supported since the earliest versions, no matter their form.

To compensate for this, we also present the signatures without the values representing these older language levels in Figure 3. Finally, in Figure 4, we present all modernity signatures, ordered by release date. Again, the results are shown with and without the earliest language levels included.

5.2 Testing Phase

Next, we show the signatures calculated in the testing phase. In Figure 5, the results are first shown grouped by project. The values corresponding to the earliest language levels have been omitted. Finally, in Figure 6, we show the results for all projects in the test phase, ordered by release date. Here, the results are displayed with and without the values for the earliest language levels.

6 DISCUSSION

6.1 Bias

In Figures 3 and 5, we see different patterns in how the signature changes over time. An unbiased signature would have a similar shape for two different systems developed in the same year, but unfortunately, this is not the case. We can thus conclude that there exists a slight bias in the signature. This can be explained by the nature of the projects in the corpus. We remark that different project authors have different strategies for maintaining their products. Compare, for example, the charts for Laravel and Joomla!, where it becomes apparent that the Joomla! team seems to strongly prefer supporting as much PHP versions as possible, while newer Laravel versions use newer PHP features.

This change in adaptation of newer language levels is probably due to the intended user base of the software. The average Joomla! user will use Joomla! on a shared web hosting environment, where they do not control the installed PHP version. Unfortunately, older versions are still actively in use today [16], and the Joomla! developers will have to account for this. On the other hand, Laravel is mostly used by developers with full access to the server on which their project is deployed, and will thus prefer to use the latest stable PHP version and the features it introduces.

6.2 Age Prediction

For most projects in the corpus, there is a correlation between the release date of a PHP system and the shape of the modernity signature. The key exception is Joomla!, but starting from the versions released after 2021, the pattern appears as well. The exact nature of the correlation differs significantly between projects. Laravel, for example, starts showing left-skewed signatures as early as 2015, while WordPress only starts behaving like this in 2020. We conclude that the signature is able to determine the relative age of a PHP system, but in its current form is not able to determine an absolute release date.

7 CONCLUSION

In this section, we answer the sub-questions set out in Section 1.3, so that we can answer the research question introduced in the same subsection.

We have given an answer to **RQ1** in Section 4. Our signature uses an AST visitor to determine the minimum language level required for the various nodes in the AST. Then, statistical analysis of the language level distribution for the various node types is applied to derive a signature using weighted sums.

Next, we answer **RQ2** in Section 6.1. There is some bias in the signature, which is due to the nature of the various software systems we used to train and test the signature. We present various opportunities for future work in Section 8, which can eliminate this bias and improve the value of the signature.

Finally, **RQ3** is answered in Section 6.2. The signature is able to determine the relative age of a system, i.e. which of two versions of the same software are newer, but not the absolute age.

In conclusion, we have shown that it is plausible that we can use grammar usage statistics to determine the modernity of a PHP system. There is room for improvement so that we will be able to determine the absolute, rather than relative, age of a system.

8 FUTURE WORK

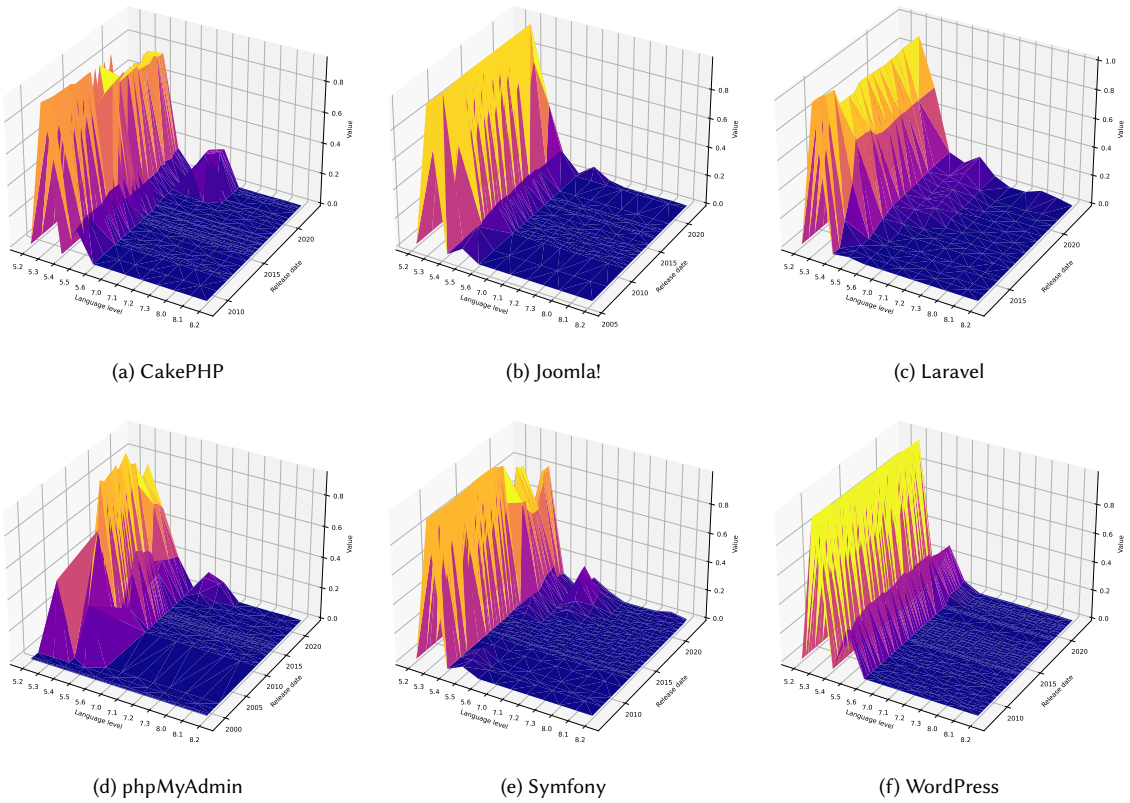
8.1 Phabricator

In the original proposal for this research, we also intended to use Phabricator[8] in the testing phase. Unfortunately, it became apparent that Phabricator does not have a versioning scheme, but rather recommends the user to download and run whatever is currently in the main branch of the Git repository. The testing and validation of the signature could be expanded with the inclusion of various snapshots of the repository.

8.2 Weighted Sum and Signature Reduction Alternatives

Currently, as described in Section 4, we use weighted sums with various weight factors to reduce the language level tuples to a single signature. This calculation method was based mostly on intuition,

[8] https://www.phacility.com/phabricator/



(a) CakePHP (b) Joomla! (c) Laravel

(d) phpMyAdmin (e) Symfony (f) WordPress

Fig. 2. Modernity signatures calculated in the testing phase, grouped by project. See also Table 2.

rather than statistical methods or previous research. One could further research various alternative methods of reducing the language level tuples in order to improve the signature.

REFERENCES

[1] George Peter Banyard. 2021. *PHP RFC: Pure intersection types.* Technical Report. The PHP Group. Retrieved May 3, 2022 from https://wiki.php.net/rfc/pure-intersection-types

[2] Peter Bumbulis and Donald D Cowan. 1993. RE2C: A more versatile scanner generator. *ACM Letters on Programming Languages and Systems (LOPLAS)* 2, 1-4 (1993), 70–84. https://doi.org/10.1145/176454.176487

[3] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2012. A survey on automated dynamic malware-analysis techniques and tools. *Comput. Surveys* 44 (2 2012), 1–42. Issue 2. https://doi.org/10.1145/2089125.2089126

[4] Larry Garfield and Ilija Tovilo. 2020. *PHP RFC: Enumerations.* Technical Report. The PHP Group. Retrieved May 3, 2022 from https://wiki.php.net/rfc/enumerations

[5] The PHP Documentation Group. 2022. PHP: History of PHP. Retrieved May 3, 2022 from https://www.php.net/manual/en/history.php.php

[6] David Hauzar and Jan Kofron. 2015. Framework for Static Analysis of PHP Applications. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 689–711. https://doi.org/10.4230/LIPIcs.ECOOP.2015.689 Keywords: Static analysis, abstract interpretation, dynamic languages, PHP, security.

[7] Mark Hills and Paul Klint. 2014. PHP AiR: Analyzing PHP systems with Rascal. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE).* IEEE, Antwerp, Belgium, 454–457. https://doi.org/10.1109/CSMR-WCRE.2014.6747217

[8] Mark Hills, Paul Klint, and Jurgen Vinju. 2013. An Empirical Study of PHP Feature Usage: A Static Analysis Perspective. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013).* Association for Computing Machinery, New York, NY, USA, 325—-335. https://doi.org/10.1145/2483760.2483786

[9] Mark Hills, Paul Klint, and Jurgen J. Vinju. 2017. Enabling PHP software engineering research in Rascal. *Science of Computer Programming* 134 (2 2017), 37–46. https://doi.org/10.1016/J.SCICO.2016.05.003

[10] JetBrains s.r.o. 2022. Code inspections | PhpStorm. Retrieved May 4, 2022 from https://www.jetbrains.com/help/phpstorm/code-inspection.html#access-inspections-and-settings

[11] Ayesh Karunaratne. 2020. *PHP's resource to object transformation.* PHP:Watch. Retrieved May 3, 2022 from https://php.watch/articles/resource-object

[12] Laurent Laville. 2022. PHP Compatinfo Home Page. Retrieved June 25, 2022 from https://llaville.github.io/php-compatinfo/6.x/

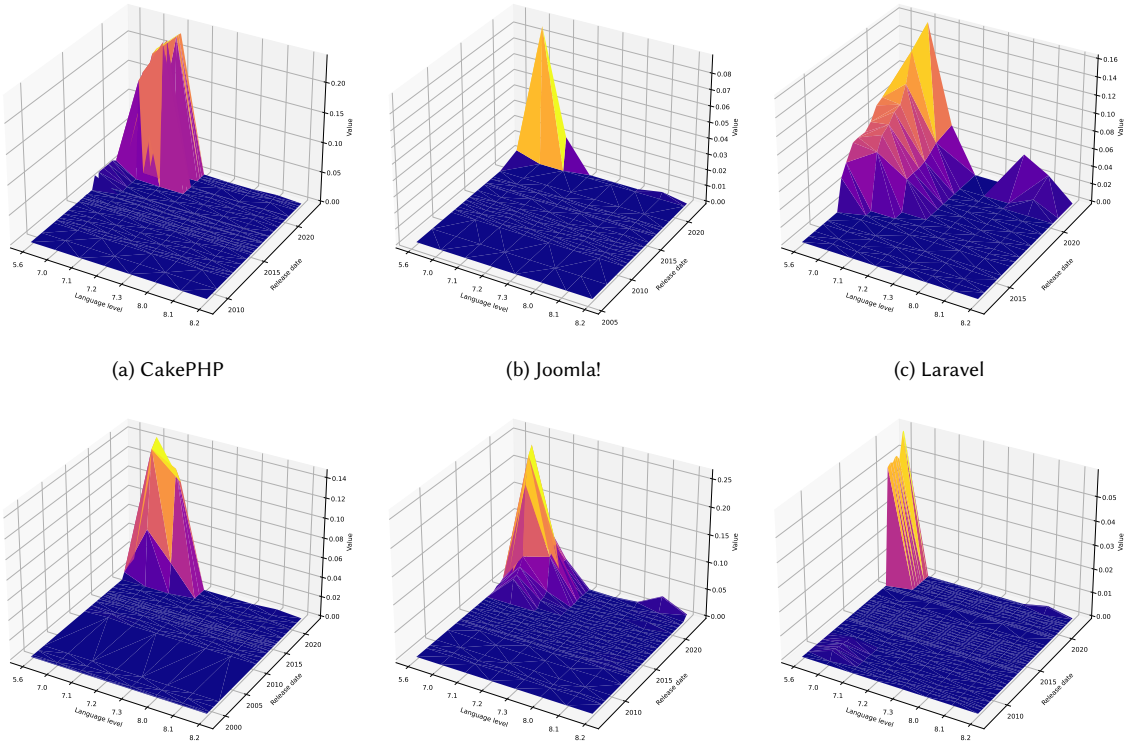
[13] Ettore Merlo, Dominic Letarte, and Giuliano Antoniol. 2007. Automated protection of php applications against SQL-injection attacks. In *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR.* IEEE Computer Society, Amsterdam, The Netherlands, 191–200. https://doi.org/10.1109/CSMR.2007.16

[14] Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. 2011. PHP Aspis: Using Partial Taint Tracking to Protect Against Injection Attacks. In *2nd USENIX Conference on Web Application Development (WebApps 11).* USENIX Association, Portland, Oregon, 13–24. https://www.usenix.org/conference/webapps11/php-aspis-using-partial-taint-tracking-protect-against-injection-attacks

[15] Tom Preston-Werner. 2013. Semantic Versioning 2.0.0. Retrieved June 25, 2022 from https://semver.org/spec/v2.0.0.html

[16] Brent Roose. 2022. PHP Version Stats: July, 2022. Retrieved July 1, 2022 from https://stitcher.io/blog/php-version-stats-july-2022

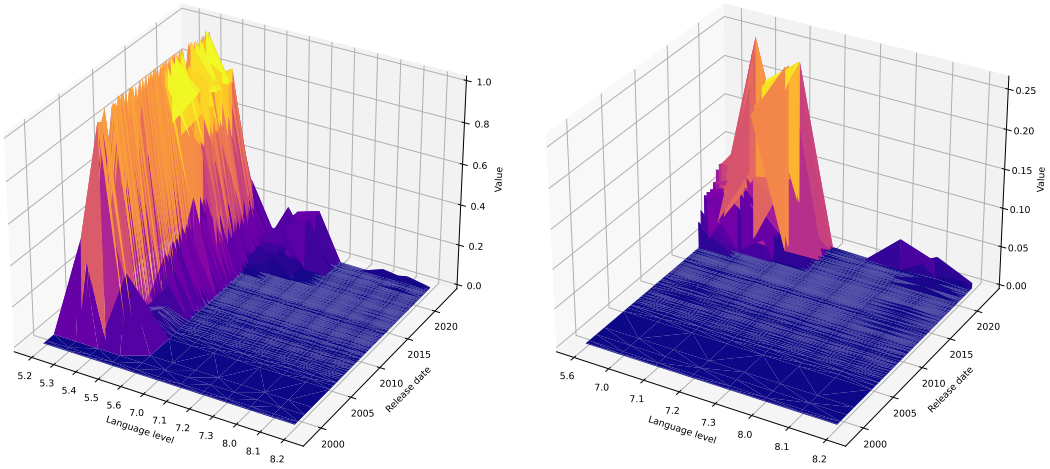
[17] Ulya Trofimovich. 2020. RE2C: A lexer generator based on lookahead-TDFA. *Software Impacts* 6 (2020), 100027. https://doi.org/10.1016/j.simpa.2020.100027



(a) CakePHP (b) Joomla! (c) Laravel

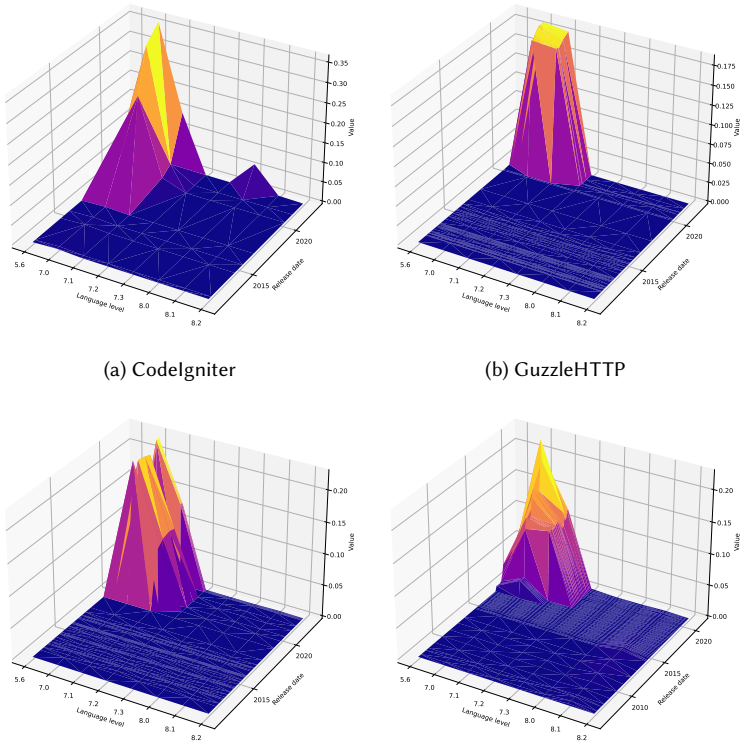
(d) phpMyAdmin (e) Symfony (f) WordPress

Fig. 3. Modernity signatures calculated in the training phase, grouped by project, without signature values for the earliest language levels. See also Table 1.



(a) With earliest language levels (b) Without earliest language levels

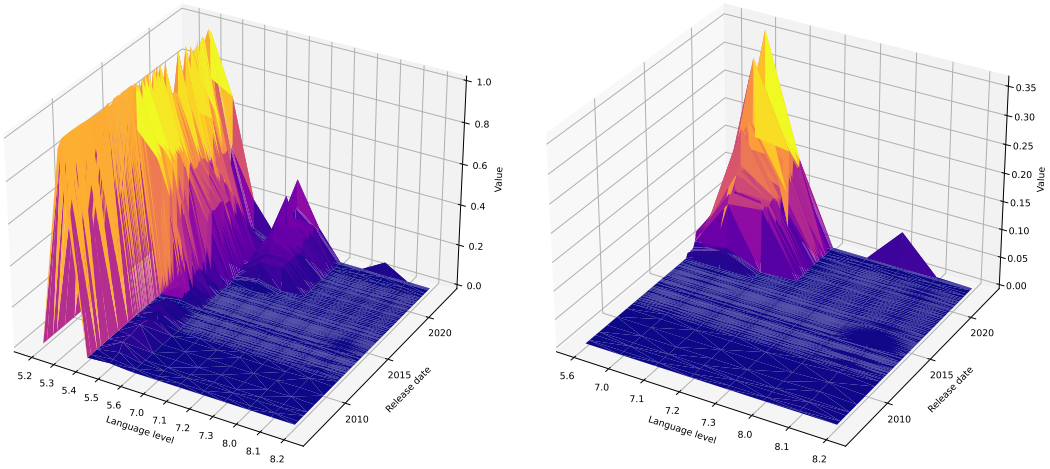
Fig. 4. Modernity signatures calculated in the training phase, with and without signature values for the earliest language levels.



(a) CodeIgniter (b) GuzzleHTTP

(c) Monolog (d) PHPUnit

Fig. 5. Modernity signatures calculated in the testing phase, grouped by project, without signature values for the earliest language levels.



(a) With earliest language levels (b) Without earliest language levels

Fig. 6. Modernity signatures calculated in the testing phase, with and without signature values for the earliest language levels.