Deriving Modernity Signatures for PHP Systems with Static Analysis

Wouter van den Brink
Technical Computer Science
University of Twente
The Netherlands
w.vandenbrink@student.utwente.nl

Marcus Gerhold
Formal Methods and Tools
University of Twente
The Netherlands
m.gerhold@utwente.nl

Vadim Zaytsev
Formal Methods and Tools
University of Twente
The Netherlands
vadim@grammarware.net

***Abstract*—The PHP language has undergone many changes in its syntax and grammar, with respect to both features the language has to offer as well as 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.**

I. Introduction

In its long history and many versions, the PHP language has undergone many changes [21]. 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 such as enumerations [4] and intersection types [1]. These syntax modifications encourage PHP programmers to use new programming paradigms in their code. Other adjustments introduced by new language versions do not change 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. There exists a continuing migration from resource types to standard class objects, further elaborated by Karunaratne [12].

A. 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 artefact produced by the Composer package manager, available at https://getcomposer.org.

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 codebase. 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 codebase 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 codebase without needing to execute the code, and without extensive human inspection.

The remainder of the paper will be spent on answering our main research question: *to what extent can we use grammar usage statistics to reliably determine the modernity of a PHP system?* We will explain our way of analysing the usage statistics in § IV, describe the corpus we used in our research in § V, report on our preliminary findings in § VI and conclude with a discussion in § VII and closing remarks in § VIII.

This is the last author's version of the work. It is posted on the author's website for your personal use, and may differ slightly layout-wise from the official published version. The definitive version was published in the Proceedings of the 22nd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), New Ideas and Emerging Results Track (NIER).

II. Motivation

Estimating the age of a codebase is a known problem in software comprehension, useful for many purposes:

- Analysing `IDENTIFICATION DIVISION`s and generated comments is one of the first steps in industrial legacy codebase analysis, in order to determine the exact language [13] and dialect [14] which define what tools are needed to handle the code.
- Age ranges are used to partition projects into new, enhancement and maintenance [10].
- For frontend systems, the age of code determines explorable vulnerabilities that it inevitably contains [17].

- For comprehension purposes, the age influences coding idioms, programming style and required expertise [3].
- For mergers and acquisitions, comparing modernities and styles of merging codebases leads to better cost estimates.
- The hazard function, failure rate, etc in information management are age-specific and, depending on the age of the project, can be negligible or overwhelming [22].
- In quantitative IT portfolio management, the age of a project helps interpreting the data about its costs(accounting for inflation and currency reforms) [22].

In this paper, we show how such process can work on PHP, but the reliance on general language processing technologies like grammars and tree traversals allows us to envision it being generalised to any other language with an available parser.

III. Related Work

With PHP being a rather dynamic language, there is a spectrum of tools for it using dynamic analysis [16], [18], static analysis or a combination thereof. Since we prefer to stay with static analysis as far as possible, the most related existing solutions for us are PhpStorm [9], PHP AiR [7], [8] and the unnamed framework by Hauzar and Kofron [6]. Most PHP code analysis tools have a strong focus on standard enforcing (e.g., smell detection) and security (including instrumentation).

In the study by Hills et al [8], statistical analysis was performed 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 analysing PHP systems to determine its age are mostly focused on determining the language level in terms of compatibility. For example, PhpStorm [9] contains a static analysis tool to determine whether language features are used which are not supported by the minimum version specified by the developer:

```
! Enumerations are only allowed since PHP 8.1 :15
! Arrow function syntax is only allowed since PHP 7.4 :55
! Arrow function syntax is only allowed since PHP 7.4 :63
```

Another example existing solution is PHP Compatinfo [15], a tool which for any given system determines the minimum required PHP version and installed extensions. The tool is mature, and gives more information on the required PHP version than PhpStorm. However, it does not give information on the age of the codebase it analyses.

In general, statistical analysis of languages and programs is a much more popular topic in natural language processing (mostly with tree adjoining grammars), but there exist attempts to use such techniques for language translation [11]. We are not aware of any projects using construct usage based modernity signatures to analyse and/or compare software systems.

IV. Methodology

We define **modernity** as a scale of measuring the age of a codebase. For the context of PHP, modernity equals language level (§ I-A), determining which version must be installed to run the code. In the context of a legacy codebase of COBOL generated from a 4GL [23], it could mean phases of 4GL's existence (e.g., "IEF" in the first comment dates the code back to 1990–1996, "Composer" to 1996–1997, "COOL:GEN" to 1997–2004, "Advantage Gen" to 2004–2012 and "CA Gen" to 2012+, while the language largely stayed the same [2]).

Our **modernity signature** takes a form of an n-tuple with n being the number of PHP language levels we consider (currently we support 13: 5.2–5.6, 7.0–7.4 and 8.0–8.2). Every element of this tuple is supposed to be a quantitative representation of the extent to which the analysed code base looks like code written to be executed by this language level.

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 analysing a PHP system, we can add annotations to the grammar of how often the different paths are taken:

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

In this example, enumerations are chosen in 7% of the cases, but enumerations were only 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.

We have used the PHP Parser from https://github.com/nikic/PHP-Parser which is the most up to date existing parser of PHP (and thus capable of dealing with all possible versions of PHP), to build our own library on. We parse the code and annotate each node of the resulting parse trees with a range (proxied by minimum and maximum for performance reasons) of language levels that support that construct. Determining these ranges was mostly straightforward since it is known which language version introduced which constructs to the language, with some very specific changes in the grammar, like syntactically preventing octal numbers from overflowing [5]. We had to get slightly creative on this step in order to declutter the signature as much as possible, up to considering disregarding counting all the features which were introduced in the oldest language version we supported (5.2). All this

designing and tweaking happened only while applying our tool to the projects from the training part of the corpus.

Each leaf of each AST gets a range of language levels/versions assigned to it, but once this data is accumulated per AST node type, the process gets somewhat more complex. Consider the following simplified example. Let us suppose that a particular node Name in a tree could only be an Identifier in one version of a language, could be either Identifier or an Expression in the second version, and only an Expression in the latest version of the language. Focusing on these three levels, our tool from six occurrences of a Name could have collected tuples $\langle 4,4,0\rangle$ for the Identifier which occurs four times across the tree, and $\langle 0,2,2\rangle$ for the Expression. We normalise both tuples such that their maximum values become 1 (and lower ones are scaled down proportionally), and calculate the resulting signature for Name as follows:

$$^4/_6 \cdot \langle 1,1,0\rangle + {}^2/_6 \cdot \langle 0,0,1\rangle = \langle {}^2/_3, 1, {}^1/_3\rangle$$

If a Name occurs in another context next to a node Type with normalised signature $\langle {}^1/_2, {}^1/_3, 1\rangle$, and there are 30 Name nodes and 20 of Type nodes within some parent context, then the signature of that context will be:

$$^{30}/_{50} \cdot \langle {}^2/_3, 1, {}^1/_3\rangle + {}^{20}/_{50} \cdot \langle {}^1/_2, {}^1/_3, 1\rangle = \langle {}^3/_5, {}^{11}/_{15}, {}^3/_5\rangle$$

This signature gets normalised as $\langle {}^9/_{11}, 1, {}^9/_{11}\rangle$, and the calculation continues. We continue traversing the nonterminals (AST node types), keeping all counters normalised such that the maximum value in each is 1, and then multiplying collected language level estimations of lower nodes by how likely such nodes were to occur according to the observed code. Eventually, the weighted sum is calculated for the entire AST. Now, the weight of the tuples is still equal to the distribution of the occurrence of the node type. Again, in this weighted sum, the tuples are normalised before the weighted sum is calculated. To calculate the signature for an entire PHP system, the weighted sum of the normalised tuples is calculated in a similar way. Then, 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 0.1 in the sum.

Our implementation is open source and can be explored at https://github.com/WoutervdBrink/PHP-Modernity-Signature, respecting the open source MIT license. We welcome forks and accept pull requests.

V. Corpus Construction

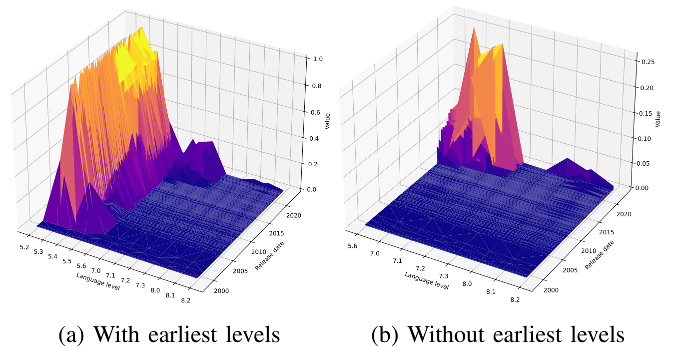
We have collected a few PHP systems to test our modernity signature, preferring well-known projects of which many historical versions were available. Our corpus consisted of:

Project	Home page	Versions
CakePHP	https://cakephp.org	29
Joomla!	https://www.joomla.org	18
Laravel	https://laravel.com	16
phpMyAdmin	https://www.phpmyadmin.net	22
Symfony	https://symfony.com	32
WordPress	https://wordpress.org	44
CodeIgniter	https://codeigniter.com	8
Guzzle HTTP	https://docs.guzzlephp.org/	37
Monolog	https://seldaek.github.io/monolog	36
PHPUnit	https://phpunit.de	57

There is no available curated corpus of PHP systems readily available for mining/evolution research. Thus, we assembled a new corpus with two parts: the training set and the testing set. All versions of all systems (299 in total) were annotated with the time of their release (either available explicitly or from the first `git` commit in the release tag) as belonging to a particular PHP level. The repositories in the top part of the table were used for developing the modernity signature that could estimate the age of the system, and visualisations obtained from them were actively used to develop and tweak the definition of the signature; and the ones on the bottom were used to verify how good those signatures work. As a matter of principle and cleanness, we have not changed the design of the modernity signature after the testing set was used.

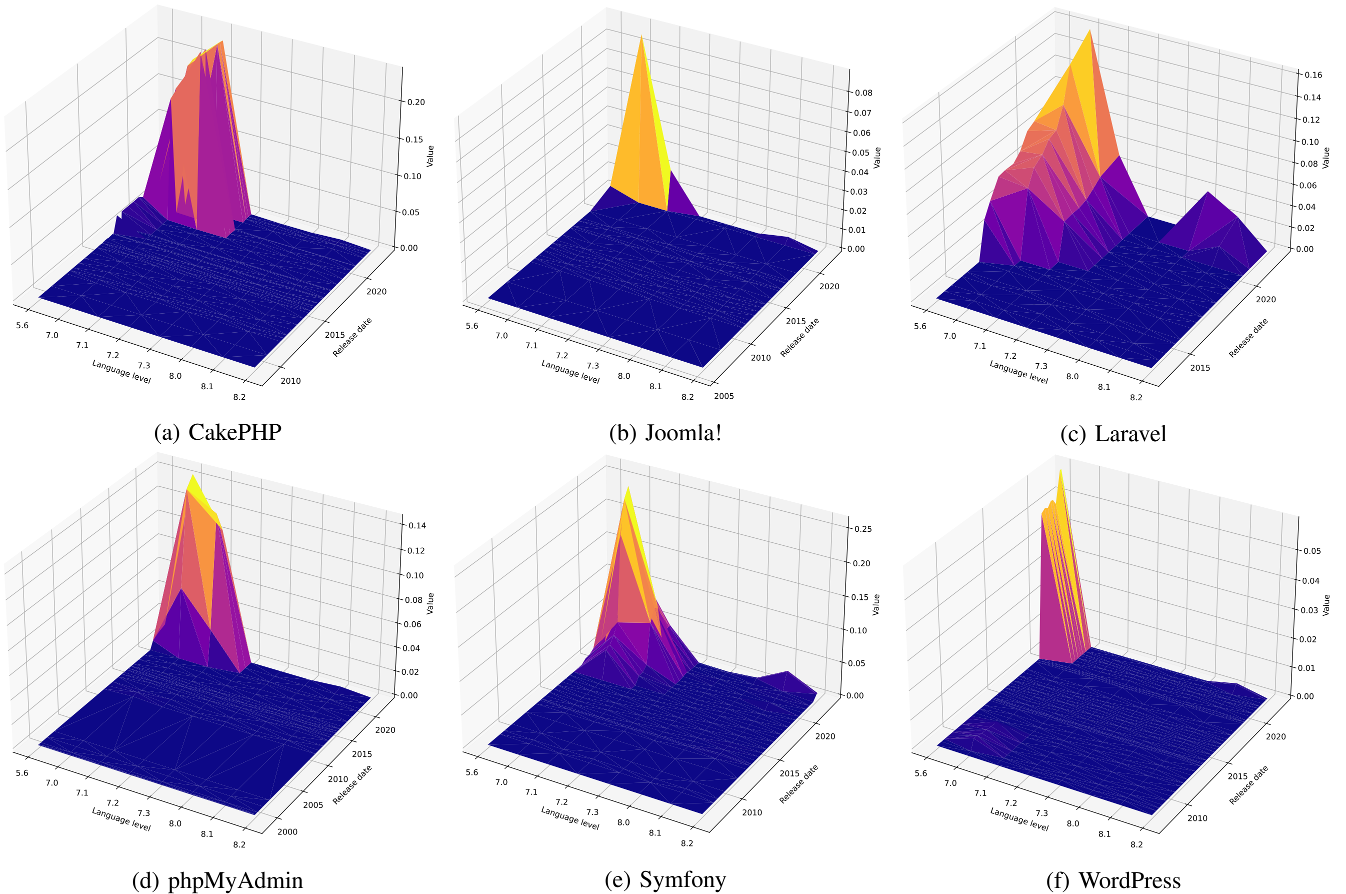
VI. Preliminary Results

The training phase showed us that the signatures tend to be heavily skewed to the right, independent of the release date of the software — accumulated results can be seen in Figure 1. This is probably due to the fact that, as explained above, most node types in the PHP language have been supported since the earliest versions, no matter their form. To compensate for this, we present the signatures without the values representing these older language levels in Figure 2. Next, we show the signatures calculated in the testing phase. In Figure 3, the results are first shown grouped by project. The values corresponding to the earliest language levels have been omitted.



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

Fig. 1: Modernity signatures in the training phase, with and without signature values for the earliest language levels.



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

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

Fig. 2: Modernity signatures calculated in the training phase, grouped by project, without signature values for the earliest language levels

VII. Discussion

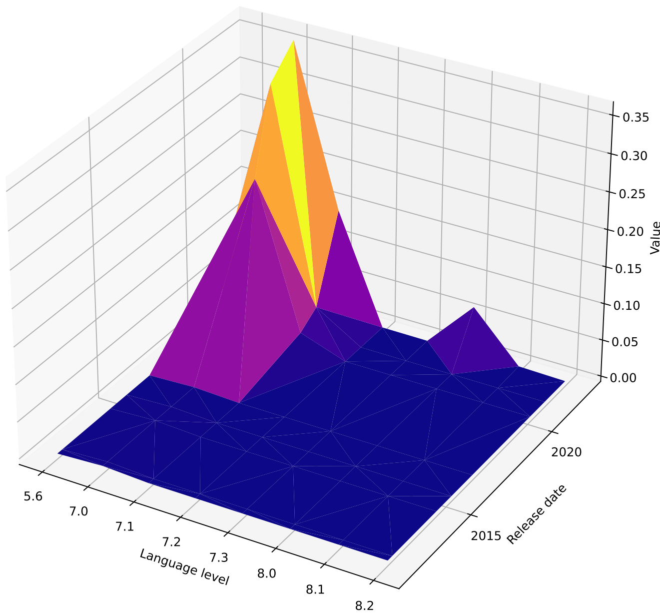
In Figures 2 and 3, we see different patterns in how the signature changes over time. An unbiased signature would have a similar shape for two different systems developed in the same year, but unfortunately, this is not the case. We can thus conclude that there exists a bias in our signature. This can be partly 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 adoption of newer language levels is explained by 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 and need to deal with dependencies on dozens of extensions (some of which are legacy on their own). Unfortunately, older versions are still actively in use today [20], and the Joomla! developers will have to account for this. On the other hand, Laravel is mostly used by "web artisans" (the term used on the main website of Laravel) 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.

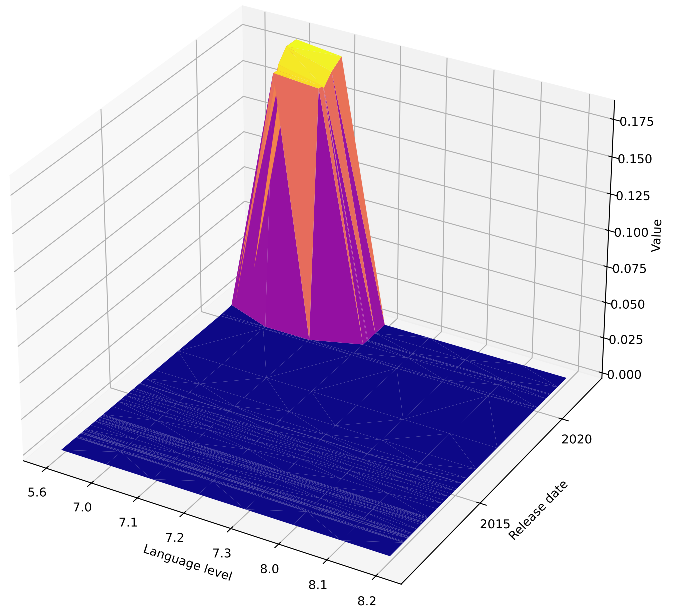
For most projects in the corpus, there is a strong connection between the release date of a PHP system and the shape of the modernity signature. The key exception is Joomla!, where this pattern appears only starting from the versions released after 2021. The exact nature of the connection differs significantly among projects. Laravel, for example, starts showing 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. It remains to be investigated how much of this observation is based (a) on the very concept of modernity based on grammar usage statistics, (b) on the peculiarities of PHP, (c) on noise and biases in our corpus.

VIII. Conclusion and Future Work

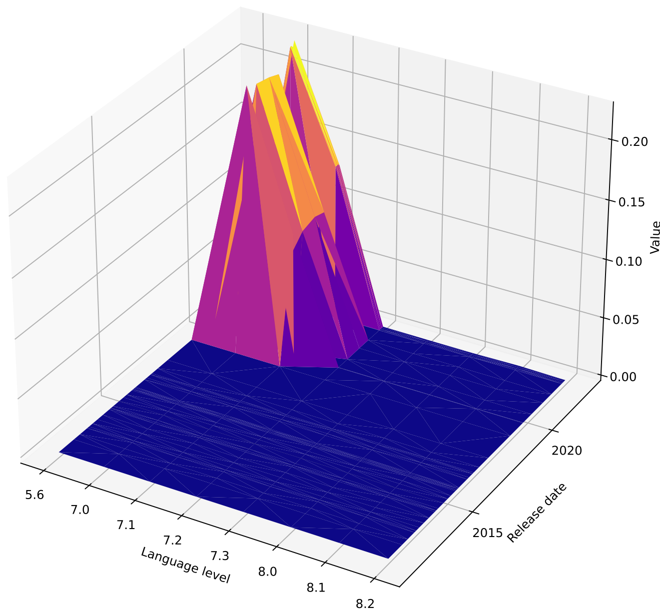
We have built a prototype for calculating modernity signatures for PHP code. Our tool uses an existing PHP parser, implements several AST visitors and uses weighted sums to converge to one n-tuple of version support estimations. The approach looks promising, and even though it displays



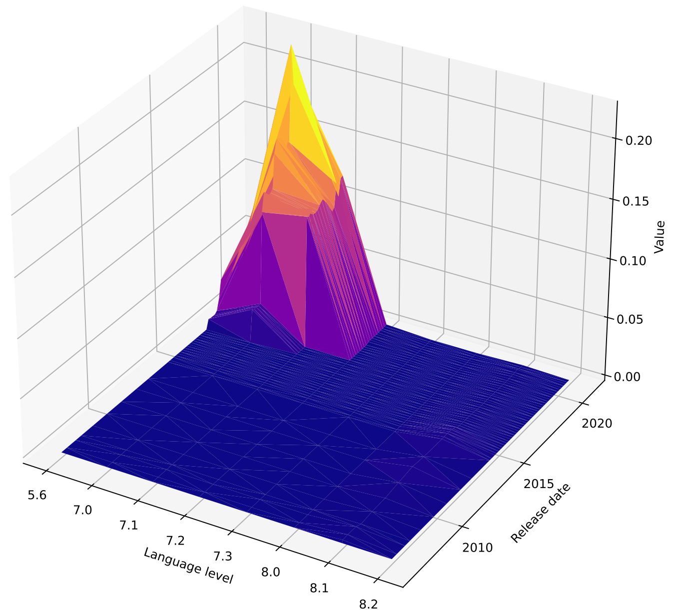
(a) CodeIgniter



(b) GuzzleHTTP



(c) Monolog



(d) PHPUnit

Fig. 3: Modernity signatures calculated in the testing phase, grouped by project, without signature values for the earliest language levels.

certain bias, that bias can be explored and exploited further. Perhaps such language usage signatures are a combination of modernity signatures and some sort of coding convention signature. It is known from prior research by Farooq et al that language feature adoption typically follows one of the three patterns: a steadily rising one (the working hypothesis for this paper as well), a one time spike (the feature gets massively adopted right after release and then massively abandoned) and a "hype curve" pattern with a slowly rising adoption, a peak of disappointment, a trough of disillusionment and an even slower second-generation adoption phase [3]. If these patterns are known per feature, it would be interesting in the future to experiment with some normalisation of the signatures to account for their shape.

In conclusion, we have shown that it is plausible to use grammar usage statistics to determine the modernity of a PHP system. At the moment our 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. There is room for improvement so that we will be able to determine

the absolute, rather than relative, age of a system.

Originally we have also intended to use Phabricator [19] 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.

Another direction to explore in the future would be to experiment with the use of weighted sums. As § IV has shown, there is a lot of normalisation and weighting already necessary to condense the knowledge obtained from codebase traversal, into a single signature, and we would rather base those weighting and normalising strategies on statistical considerations than on intuition. One could further research various alternative methods of reducing the language level tuples in order to improve the signature.

References

[1] G. P. Banyard, "PHP RFC: Pure intersection types," The PHP Group, Tech. Rep., 2021. [Online]. Available: https://wiki.php.net/rfc/pure-intersection-types

[2] CA Technologies, "5 Ways DevOps Practices Boost Innovation on the Mainframe," CS 200-227965, https://docs.broadcom.com/doc/5-ways-devops-practices-boost-innovation-on-the-mainframe, 2016.

[3] A. Farooq and V. Zaytsev, "There Is More Than One Way to Zen Your Python," in *Proceedings of the 14th International Conference on Software Language Engineering (SLE)*, E. Visser, D. Kolovos, and E. Söderberg, Eds. ACM, 2021, pp. 68–82. [Online]. Available: https://doi.org/10.1145/3486608.3486909

[4] L. Garfield and I. Tovilo, "PHP RFC: Enumerations," The PHP Group, Tech. Rep., 2020. [Online]. Available: https://wiki.php.net/rfc/enumerations

[5] S. Golemon, "PHP RFC: Fix Overflow in Octal Parsing," The PHP Group, Tech. Rep., 2016. [Online]. Available: https://wiki.php.net/rfc/octal.overload-checking

[6] D. Hauzar and J. Kofron, "Framework for Static Analysis of PHP Applications," in *Proceedings of the 29th European Conference on Object-Oriented Programming*, ser. Leibniz International Proceedings in Informatics, vol. 37. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2015, pp. 689–711. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2015.689

[7] M. Hills and P. Klint, "PHP AiR: Analyzing PHP systems with Rascal," in *Proceedings of the Software Evolution Week: Conference on Software Maintenance, Reengineering, and Reverse Engineering*, S. Demeyer, D. Binkley, and F. Ricca, Eds. IEEE Computer Society, 2014, pp. 454–457. [Online]. Available: https://doi.org/10.1109/CSMR-WCRE.2014.6747217

[8] M. Hills, P. Klint, and J. J. Vinju, "Enabling PHP Software Engineering Research in Rascal," *Science of Computer Programming*, no. 134, pp. 37–46, 2017. [Online]. Available: https://doi.org/10.1016/J.SCICO.2016.05.003

[9] JetBrains, "Code inspections — PhpStorm," https://www.jetbrains.com/help/phpstorm/code-inspection.html, 2022.

[10] C. Jones, "Patterns of Large Software Systems: Failure and Success," *Computer*, vol. 28, no. 3, pp. 86–87, 1995.

[11] S. Karaivanov, V. Raychev, and M. T. Vechev, "Phrase-Based Statistical Translation of Programming Languages," in *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, A. P. Black, S. Krishnamurthi, B. Bruegge, and J. N. Ruskiewicz, Eds. ACM, 2014, pp. 173–184. [Online]. Available: https://doi.org/10.1145/2661136.2661148

[12] A. Karunaratne, "PHP's Resource to Object Transformation," PHP:Watch, https://php.watch/articles/resource-object, 2020.

[13] J. Kennedy van Dam and V. Zaytsev, "Software Language Identification with Natural Language Classifiers," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering: the Early Research Achievements track (SANER ERA)*, K. Inoue, Y. Kamei, M. Lanza, and N. Yoshida, Eds. IEEE, 2016, pp. 624–628. [Online]. Available: https://doi.org/10.1109/SANER.2016.92

[14] R. Lämmel and C. Verhoef, "Cracking the 500-language problem," *IEEE Software*, vol. 18, no. 6, pp. 78–88, 2001. [Online]. Available: https://doi.org/10.1109/52.965809

[15] L. Laville, "PHP Compatinfo Home Page," https://llaville.github.io/php-compatinfo/6.x/, 2022.

[16] E. Merlo, D. Letarte, and G. Antoniol, "Automated Protection of PHP Applications Against SQL-injection Attacks," in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, R. L. Krikhaar, C. Verhoef, and G. A. Di Lucca, Eds. IEEE Computer Society, 2007, pp. 191–202. [Online]. Available: https://doi.org/10.1109/CSMR.2007.16

[17] I. Muscat, "Web Vulnerabilities: Identifying Patterns and Remedies," *Network Security*, vol. 2016, no. 2, pp. 5–10, 2016. [Online]. Available: https://doi.org/10.1016/S1353-4858(16)30016-2

[18] I. Papagiannis, M. Migliavacca, and P. R. Pietzuch, "PHP aspis: Using partial taint tracking to protect against injection attacks," in *Proceedings of the Second USENIX Conference on Web Application Development (WebApps)*, A. Fox, Ed. USENIX Association, 2011. [Online]. Available: https://www.usenix.org/conference/webapps11/php-aspis-using-partial-taint-tracking-protect-against-injection-attacks

[19] E. Priestley, "Phacility — Phabricator," https://www.phacility.com/phabricator/, 2010.

[20] B. Roose, "Php version stats: July, 2022," https://stitcher.io/blog/php-version-stats-july-2022, 6 2022.

[21] The PHP Documentation Group, "PHP: History of PHP," https://www.php.net/manual/en/history.php, 2022.

[22] C. Verhoef, "Quantitative IT portfolio management," *Science of Computer Programming*, vol. 45, no. 1, pp. 1–96, 2002. [Online]. Available: https://doi.org/10.1016/S0167-6423(02)00106-5

[23] V. Zaytsev and J. Fabry, "Fourth Generation Languages are Technical Debt," in *International Conference on Technical Debt*, 2019, Extended Abstract, http://grammarware.net/text/2019/4gl-techdebt.pdf.