Enabling PHP Software Engineering Research in Rascal

Mark Hills[a], Paul Klint[a,b]

[a] *Centrum Wiskunde & Informatica, Amsterdam, The Netherlands*
[b] *INRIA Lille Nord Europe, Lille, France*

Abstract

Today, PHP is one of the most popular programming languages and is commonly used in the open source community and in industry to build large application frameworks and web applications. In this paper, we discuss our ongoing work on PHP AiR, a framework for PHP Analysis in Rascal. PHP AiR is focused especially on program analysis and empirical software engineering, and is being used actively and effectively in work on evaluating PHP feature usage, program analysis for refactoring and security validation, and source code metrics. We describe the requirements and design decisions for PHP AiR, summarize current research using PHP AiR, discuss lessons learned, and briefly sketch future work.

Keywords: meta-programming, program analysis, empirical software engineering, dynamic languages, PHP

1. Introduction

PHP,[1] invented by Rasmus Lerdorf in 1994, is an imperative, object-oriented language focused on server-side application development. It is now one of the most popular languages, as of April 2013 ranking 6th on the TIOBE programming community index,[2] used by 78.8 percent of all websites whose server-side language can be determined,[3] and ranking as the 6th most popular language on GitHub.[4] This popularity has led to the creation of a number of large, widely-used open source applications and application frameworks, including WordPress,[5] Joomla,[6] Drupal,[7] MediaWiki,[8] Symfony,[9] and CodeIgniter.[10]

Email addresses: Mark.Hills@cwi.nl (Mark Hills), Paul.Klint@cwi.nl (Paul Klint)

[1] http://www.php.net
[2] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[3] http://w3techs.com/technologies/details/pl-php/all/all
[4] https://github.com/languages/PHP
[5] http://wordpress.org/
[6] http://www.joomla.org/
[7] http://drupal.org/
[8] http://www.mediawiki.org/wiki/MediaWiki
[9] http://symfony.com/
[10] http://ellislab.com/codeigniter

The availability of such large, open-source systems provides an ideal ecosystem for empirical software engineering research. PHP is also a fascinating subject for program analysis research. Most PHP applications are web-based, giving an urgency to program analysis targeted at detecting potential security errors. At the same time, the dynamic nature of the language (e.g., duck typing, reflection, evaluation of code built at runtime using strings), as well as its use on larger and larger systems, increases the importance of analyses targeted at program understanding, automated code refactoring, and programmer tool support, all areas where PHP currently lags behind languages such as Java.

To enable research in program analysis, automated refactoring, tool support, and empirical software engineering in PHP, we are developing PHP AiR, an environment for PHP Analysis in Rascal. Built using the Rascal meta-programming language [1], a successor to ASF+SDF [2] and RSCRIPT [3], PHP AiR has been, and is currently being, used in multiple research projects: an empirical survey of PHP language feature usage [4], a program analysis for resolving dynamic file includes [5], a taint analysis for detecting dangerous uses of unchecked user-provided strings in PHP library calls, a refactoring from hand-coded HTML to uses of template libraries, a similar refactoring from hand-coded SQL calls to uses of database libraries, and multiple projects to extract various metrics from PHP source code.

The rest of this paper is organized as follows. In Section 2, we provide a brief introduction to Rascal. Section 3 constitutes the core of our paper, discussing the requirements and design decisions for PHP AiR, giving a high-level introduction to the tool, and presenting some of the research performed so far, which helps to motivate these decisions and indicates how they have worked in practice. Section 4 presents related work for Rascal and for the analysis of PHP programs, while Section 5 concludes, discussing lessons learned and future directions. The PHP AiR system is available online at `https://github.com/cwi-swat/php-analysis`, while more information about Rascal is available at `http://www.rascal-mpl.org/`.

2. Rascal

Rascal was designed to cover the entire domain of meta-programming. The language itself is designed with unofficial "language layers." This allows inexperienced Rascal programmers to start with just the core language features, adding more advanced features as they become more comfortable with the language. This language core contains basic data-types for booleans, integers, reals, source locations, date-time, lists, sets, tuples, maps, and relations; structured control flow, e.g., if, while, switch, for; and exception handling with try and catch. The syntax of these constructs is designed to be familiar to programmers: for instance, if statements and try/catch blocks look like those found in C and Java, respectively. All data in Rascal is immutable (i.e., no references are ever created or taken), and all code is statically typed. The built-in data type of source locations is particularly suited for creating references to source fragments as they appear during analysis and transformation of source code.

Rascal's type system is organized as a lattice, with bottom (`void`) and top (`value`) elements. The Rascal `node` type is the parent of all user-defined datatypes, including the types of concrete syntax elements (e.g., `Stmt` and `Expr`). Numeric types also have a parent type, `num`, but are not themselves in a subtype relation (e.g., `real` is not a parent of `int`).

Beyond the type system and the language core, Rascal also includes a number of more advanced features. These features can be progressively used by the programmer to create more complex programs, and are needed in Rascal to enable the full range of meta-programming capabilities. These more advanced features include parameterized algebraic data type definitions; a built-in grammar formalism, which includes disambiguation facilities and annotatable grammar rules and is used to generate scannerless generalized top-down parsers; pattern matching over all Rascal types, including set matching, list matching, and deep matching (i.e., matching at an arbitrary depth within a term) over nested structures, as well as pattern-based dispatch for invoking Rascal functions; comprehensions over lists, sets, and maps; `visit` statements for performing structure-shy traversals (allowing the visit to match just those cases of interest, with default traversal behavior for the rest) and transformation of Rascal terms; powerful string templating capabilities; and a built-in notion of fixpoint computation. Rascal resources [6] provide access to external sources of data from within Rascal, leveraging the Rascal type system to ensure that uses of external data are well-typed and to provide more convenient access (e.g., by providing field names based on column names in a database table).

A number of Rascal features focus on the safety and modularity of Rascal code. While local variable types can be inferred, parameter and return types in functions must be provided. This allows better error messages to be generated, since errors detected by the inferencer can be localized within a function, and also provides documentation (through type annotations) on function signatures. Also, the only casting mechanism is pattern matching, which prevents the problems with casts found in C (lack of safety) and Java (runtime casting exceptions). Finally, the use of persistent data structures eliminates a number of standard problems with using references which can leak out of the current scope or be captured by other variables.

3. PHP AiR: PHP Analysis in Rascal

PHP AiR is being built with certain high-level requirements in mind, and a number of design decisions have been made during development of the tool. Below we discuss these requirements and design decisions, provide a high-level overview of the tool, and discuss ongoing research using PHP AiR in the domains of empirical software engineering and program analysis.

3.1. Requirements

When building PHP AiR we had several core requirements. First, and most importantly, it should be possible to use PHP AiR to effectively and efficiently

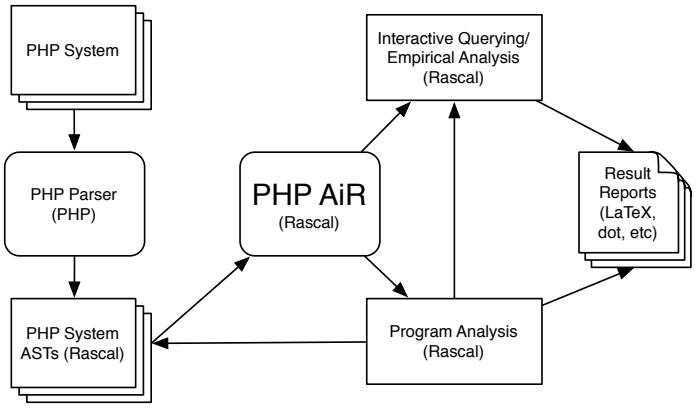

Figure 1: High-Level Overview: PHP AiR.

support empirical software engineering and program analysis research on real PHP systems, even large systems such as MediaWiki, with over 845K lines of code as of version 1.19.1. Second, PHP AiR should be interactive to enable what-if analyses and exploratory programming. For instance, it should be possible to write queries over PHP code to find all uses of a given language feature, or to prototype new program analysis tools. Third, PHP AiR should support integration with a standard PHP development environment, allowing it to be used, in a familiar way, by the largest possible audience. Finally, given that Rascal has been developed in our group for exactly these kinds of applications, we wanted PHP AiR to profit as much as possible from Rascal's language features and libraries while not excluding the use of external tools (e.g., Eclipse-based PHP tools). These external tools are made accessible by creating Rascal bindings to existing Java libraries for interacting with these tools whenever that would be advantageous. At the same time, the development of PHP AiR gives valuable feedback to the Rascal development team and a strong incentive to address any issues that are being raised, some of which are discussed below.

3.2. Design Decisions

PHP AiR has been developed completely in Rascal except for the parser, which is written in PHP itself, and currently consists of almost 12,000 lines of Rascal code and (in the parser) 1430 lines of PHP code. Figure 1 gives a high-level overview of PHP AiR. Individual PHP files, or whole systems (e.g., WordPress), are parsed and converted into Rascal terms representing ASTs. These ASTs are then the base structure over which all other operations in PHP AiR are built. Operations in PHP AiR are divided into two general categories (which, in reality, can sometimes overlap): interactive querying and empirical analysis on the one hand, program analysis on the other. In the first, the user, using Rascal, can make a number of queries over the PHP system, supplemented by external sources of data, and can also script these queries, aggregate results,

and use various standard Rascal functions for statistical analysis. Results are computed and then written either to the console or to an external format, such as a table or figure in a LaTeX document or a `.dot` file representing a graph. In the second, users can run either predefined or their own program analysis passes. The analysis results can be used to transform existing programs in the system, to supplement interactive queries or empirical analysis, or to display results. We are working on extending this display capability to take advantage of Eclipse, for instance by allowing warnings or errors computed by an analysis to be flagged directly in the PHP source files.

The design of PHP AiR, leading to this functionality, has involved a number of trade-offs. The first design decision related to parsing: should we use the built-in Rascal parsing functionality, or reuse an existing parser? As mentioned above, we decided early on to select the second option. While Rascal provides powerful parsing capabilities, building a PHP parser from scratch would be a significant undertaking. Instead, we parse PHP scripts using our fork[11] of an open-source PHP parser[12] which generates Rascal terms representing the ASTs of PHP programs. This provided a quicker start and makes it possible to take advantage of new PHP language constructs as they are added to the parser.

One downside of this approach is that it does not support interacting with PHP code in an IDE. To provide IDE support, we are building an integration layer between PHP AiR and the Eclipse PHP Developer Tools (PDT), similar to the integration that Rascal currently has with the Eclipse Java Developer Tools (JDT), which we have been able to exploit in areas such as refactoring [7]. Both approaches to parsing PHP target the same abstract syntax, allowing either to be used by the other tools developed in PHP AiR.

A second decision, which confronted us when we started developing the initial analysis passes, was: should we use optimized data structures written in Java, or use the standard Rascal data types? The second provides cleaner code, while the first could provide improved performance when analyzing large systems. Here, we actually started with the second, moved to the first, and have now returned to the second. Originally the built-in data types proved to use too much memory, motivating us to switch to using optimized data types written in Java and available through Rascal libraries. However, what we found is that the algorithms are much more important to performance than the data structures used to store analysis information as the computation proceeds, which has led us to switch back to using the built-in Rascal types while also looking at faster algorithms and improvements to the datatype implementations.

A third decision was to provide support, within Rascal, for accessing the types of external data needed for PHP AiR. This decision led to our ongoing research on Rascal resources [6], mentioned above in Section 2. In cases where optimized storage is needed for extracted facts, this should allow us to store these facts in databases while, at the same time, providing a native Rascal interface

[11] `https://github.com/cwi-swat/PHP-Parser`
[12] `https://github.com/nikic/PHP-Parser/`

to read and write these facts. We currently support read-only resources, but are working on extending this work to support writing to resources as well as more intelligent methods of reading from resources that will minimize memory use.

3.3. Empirical Software Engineering

Our current research has focused on using empirical software engineering techniques [8] to examine feature usage in large open-source PHP systems [4]. To start, we assembled a corpus of 19 large open-source PHP systems, basing our choice on popularity rankings provided by Ohloh,[13] a site that tracks open-source projects. In total, the corpus consists of 19,816 PHP source files with 3,370,219 lines of PHP source (counted using the cloc[14] tool). An extension to this corpus, looking specifically at dynamic file includes, adds 20 additional systems selected from the GitHub PHP page,[15] specifically from the most starred and most forked repositories for the day, week, month, and overall, with the goal of ensuring that more "regular" code (i.e., not just large, well-maintained projects) was also covered. This added an additional 15,492 files with 1,805,333 lines of PHP source.

Using this corpus, we focused on both general characteristics of PHP programs, such as the size of PHP files and the distribution of PHP language features, as well as on the use of dynamic features in the code. For the latter, we looked at how often these features occur in practice, how distributed these features are in PHP programs, and also how often these dynamic features are actually static in practice, meaning that static techniques can be used to reason about these features and minimize their impact on program analysis tools. Interesting findings include that `eval` is rarely used in practice (only 148 times in total in the original corpus), that variable variables (variables that contain the name of another variable, allowing indirect access) can, in many cases, be resolved statically to a specific set of referenced variables, and that many new features such as `goto` and traits are not yet used in popular systems (no uses at all were found in the corpus for either feature). At the same time, ongoing student projects and industry collaborations are using similar techniques to derive metrics from PHP code.

This work has led to performance improvements in Rascal to better handle large quantities of PHP system data, and has helped us improve the built-in statistics libraries. It has also helped us improve PHP AiR to support easier querying over language constructs as well as simpler ways of aggregating results to present online summaries and generate LaTeX tables and figures.

3.4. Program Analysis

Our ongoing work on program analysis using PHP AiR is focused on a number of analysis tasks common to other programming languages as well as

[13] `http://www.ohloh.net/tags/php`
[14] `http://cloc.sourceforge.net`
[15] `https://github.com/languages/PHP`

several that are more specific to PHP. For the first, we are building type inference and alias analysis passes that will provide information useful for programmer tools and other analysis passes. For the second, we are working on a taint analysis to detect possible security violations in calls to system functions (e.g., uses of unchecked user input to construct database queries), and have already implemented an analysis to determine, in many cases, which files are actually included by dynamic file inclusion expressions [5]. We are also working on the analyses needed to handle various refactoring operations, including a string analysis that will be needed to convert uses of HTML and SQL strings, built using string-building operations, into safer uses of HTML and SQL libraries.

Although some of the challenges we faced are based on the dynamic nature of PHP, other challenges are caused by the size of the systems we are analyzing, along with the fact that Rascal is not yet optimized for memory consumption. Practically, this means that we need to focus quite heavily on efficient, modular algorithms, and are also continuing to improve the memory footprint and overall performance of Rascal. On the other hand, features of Rascal such as source locations have proven to be quite valuable, providing a way to easily tie back error information derived from ASTs to specific points in the source code. The availability of data types for maps and relations has also provided natural ways to represent many of the facts needed during program analysis.

4. Related Work

The design of Rascal is based on inspiration from many earlier languages and systems. The syntax features (grammar definition and parsing) are directly based on SDF [2], but the notation has changed and the expressivity has increased (c.f., earlier work discusses this evolution [9]). The features related to analysis are mostly based on relational calculus, relational algebra and logic programming systems such as Crocopat [10], Grok [11] and RScript [3], with some influence from CodeSurfer [12]. Rascal has strongly simplified backtracking and fixed point computation features reminiscent of constraint programming and logic programming systems like Moreau's Choice Point Library[13], Prolog and Datalog [14]. Rascal's program transformation and manipulation features are directly inspired by term rewriting/functional languages such as ASF+SDF [15], Stratego [16], TOM [17], and TXL [18]. The ATerm library [19] inspired Rascal's immutable values, while the ANTLR tool-set [20], Eclipse IMP [21] and TOM [17] have been an inspiration because of their integration with mainstream programming environments.

A number of tools have been developed for the analysis of PHP programs. The PHP-sat[16] and PHP-tools[17] projects include limited support (mainly intraprocedural) for security analysis as well as analyses to detect a variety of

[16] `http://www.program-transformation.org/PHP/PhpSat`
[17] `http://www.program-transformation.org/PHP/PhpTools`

common bug patterns (e.g., assigning the result of a function that does not contain a return statement). PHP AiR is targeting more complex PHP programs and a wider variety of analyses. More focused tools include PHP_CodeSniffer,[18] which checks PHP code for violations of defined coding standard, and the PHP Copy/Paste Detector,[19] which provides a very limited form of clone detection (i.e., exact textual copies only). There are also several tools for calculating metrics for PHP code, including PHPDepend[20] and PHPLoc.[21] We are integrating in similar functionality, with hopefully better performance—PHPLoc is fast, but gives limited information, while PHPDepend is more complete but is quite slow when run on larger codebases. PHPMD[22] both computes metrics and tries to find a number of programming flaws and potential bugs, but focuses mainly on areas that do not require sophisticated analysis. Biggar [22] with `phc` and Zhao et al. [23] with `HipHop` both perform analysis (under various assumptions about the code) as part of the task of compiling PHP code, while Huang et al's WebSSARI [24] and Jovanovic et al.'s Pixy system [25, 26] use a combination of static analysis and (in the case of WebSSARI) program instrumentation to protect against security vulnerabilities. While we are providing an environment with PHP AiR where similar analyses can be created, we are also looking at a number of novel analyses, including one to detect bugs introduced by changes to the semantics of PHP which occur as the language continues to evolve.

5. Lessons Learned and Future Directions

From our experience building PHP AiR we have been able to extract a number of important lessons:

- Rascal's high-level data types (e.g., sets, maps, relations, and ADTs) and language features (pattern matching, traversal, local backtracking, comprehensions) all favor a declarative programming style where the distance between a published algorithm and its Rascal implementation is often surprisingly small. This enables easy experimentation at the algorithmic level. It also leads to a significantly smaller code size for the resulting tools compared to similar tools implemented in more traditional programming languages.

- Location information is important for research looking at language features and critical for program analysis, where it can be used to provide accurate messages. As already mentioned earlier, Rascal includes a source location data type as a language feature, with locations being used extensively in the libraries for IDE integration (e.g., for error reporting and

[18] `http://pear.php.net/package/PHP_CodeSniffer`
[19] `https://github.com/sebastianbergmann/phpcpd`
[20] `http://pdepend.org/`
[21] `https://github.com/sebastianbergmann/phploc`
[22] `http://phpmd.org/`

source code annotation). This simplifies the processing of references to source code fragments.

- Flexibility in secondary tool choices, such as parsing, has been key to quickly getting PHP AiR off the ground. Since all code in PHP AiR works over AST nodes supplemented with location information, different parsers can be used as front-ends without needing broader changes to the source code of PHP AiR itself.

- The ability to script empirical analyses, and even to script the generation of artifacts for research papers such as tables and figures, has been an important feature, allowing us to make the results of our research reproducible in a form that can be more easily checked.

- The availability of Rascal resources has provided a clean way to reuse data created by external tools, and should be extended to encompass additional data sources.

- Performance is a persistent issue, especially when analyzing large systems, and needs to be addressed in PHP AiR and directly within Rascal. There are still a number of cases where memory use or execution performance are unsatisfactory, but, more positively, these cases are driving improvements in the algorithms used in PHP AiR and in the implementation of Rascal.

We have a number of future plans for further developing and utilizing PHP AiR. First, we plan to complete integration with the Eclipse PHP Development Tools, allowing analysis and transformation of the code to be directed from within Eclipse. Second, we plan to continue making improvements to the performance of Rascal to allow us to handle larger codebases. Third, we also plan to continue work on program analysis and empirical software engineering using PHP AiR, which is proving to also be a good environment for student projects.

In the longer term, we plan to utilize persistent storage to more easily store and process the intermediate results of our analyses. Also, to raise the level of abstraction in creating the various components of PHP AiR, we are working on domain-specific languages for tasks such as intermediate language generation and control flow graph construction. These DSLs should be usable for creating tools for languages beyond PHP.

References

[1] P. Klint, T. van der Storm, J. J. Vinju, RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation, in: Proceedings of SCAM'09, IEEE, 2009, pp. 168–177.

[2] J. Heering, P. Hendriks, P. Klint, J. Rekers, The syntax definition formalism SDF - reference manual, SIGPLAN Notices 24 (11) (1989) 43–75.

[3] P. Klint, Using Rscript for Software Analysis, in: Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008), 2008.

[4] M. Hills, P. Klint, J. J. Vinju, An Empirical Study of PHP Feature Usage: A Static Analysis Perspective, in: Proceedings of ISSTA'13, ACM Press, 2013, to appear, available at `http://homepages.cwi.nl/~hills/publications/php-feature-usage.pdf`.

[5] M. Hills, P. Klint, J. J. Vinju, Statically Resolving Dynamic Includes in PHP, preprint available at `http://homepages.cwi.nl/~hills/publications/resolving-php-includes.pdf`.

[6] M. Hills, P. Klint, J. J. Vinju, Meta-language Support for Type-Safe Access to External Resources, in: Proceedings of SLE'12, Vol. 7745 of LNCS, Springer, 2012, pp. 372–391.

[7] M. Hills, P. Klint, J. J. Vinju, Scripting a Refactoring with Rascal and Eclipse, in: Proceedings of WRT'12, ACM, 2012, pp. 40–49.

[8] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, Experimentation in Software Engineering, Springer, 2012.

[9] J. van den Bos, M. Hills, P. Klint, T. van der Storm, J. J. Vinju, Rascal: From Algebraic Specification to Meta-Programming, in: Proceedings of AMMSE'11, Vol. 56 of EPTCS, 2011, pp. 15–32.

[10] D. Beyer, Relational programming with CrocoPat, in: Proceedings of ICSE'06, ACM Press, 2006, pp. 807–810.

[11] R. C. Holt, Grokking Software Architecture, in: Proceedings of WCRE'08, IEEE, 2008, pp. 5–14.

[12] P. Anderson, M. Zarins, The CodeSurfer Software Understanding Platform, in: Proceedings of IWPC'05, IEEE, 2005, pp. 147–148.

[13] P.-E. Moreau, A choice-point library for backtrack programming, JICSLP'98 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic (1998).

[14] S. Ceri, G. Gottlob, L. Tanca, Logic programming and databases, Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[15] M. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, J. Visser, The ASF+SDF Meta-environment: A Component-Based Language Development Environment, in: Proceedings of CC'01, Vol. 2027 of LNCS, Springer, 2001, pp. 365–370.

[16] M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.17. A Language and Toolset for Program Transformation, Science of Computer Programming 72 (1-2) (2008) 52–70.

[17] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, A. Reilles, Tom: Piggybacking Rewriting on Java, in: Proceedings of RTA'07, Vol. 4533 of LNCS, Springer, 2007, pp. 36–47.

[18] J. R. Cordy, The TXL source transformation language, Science of Computer Programming 61 (3) (2006) 190–210.

[19] M. van den Brand, H. de Jong, P. Klint, P. Olivier, Efficient Annotated Terms, Software, Practice & Experience 30 (2000) 259–291.

[20] T. Parr, The Definitive ANTLR Reference: Building Domain-Specific Languages, Pragmatic Bookshelf, 2007.

[21] P. Charles, R. M. Fuhrer, S. M. Sutton Jr., E. Duesterwald, J. J. Vinju, Accelerating the Creation of Customized, Language-Specific IDEs in Eclipse, in: Proceedings of OOPSLA'09, ACM Press, 2009, pp. 191–206.

[22] P. Biggar, Design and Implementation of an Ahead-of-Time Compiler for PHP, Ph.D. thesis, Trinity College Dublin (April 2010).

[23] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans, S. Tu, The HipHop Compiler for PHP, in: Proceedings of OOPSLA'12, ACM, 2012, pp. 575–586.

[24] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, S.-Y. Kuo, Securing Web Application Code by Static Analysis and Runtime Protection, in: Proceedings of WWW'04, ACM, 2004, pp. 40–52.

[25] N. Jovanovic, C. Krügel, E. Kirda, Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper), in: IEEE Symposium on Security and Privacy, 2006, pp. 258–263.

[26] N. Jovanovic, C. Kruegel, E. Kirda, Precise Alias Analysis for Static Detection of Web Application Vulnerabilities, in: Proceedings of PLAS'06, ACM, 2006, pp. 27–36.