Toward an Automated Benchmark Management System

Lisa Nguyen Quang Do
Fraunhofer IEM, Germany
lisa.nguyen@iem.fraunhofer.de

Michael Eichberg
Technische Universität Darmstadt, Germany
eichberg@informatik.tu-darmstadt.de

Eric Bodden
Paderborn University & Fraunhofer IEM, Germany
eric.bodden@upb.de

Abstract

The systematic evaluation of program analyses as well as software-engineering tools requires benchmark suites that are representative of real-world projects in the domains for which the tools or analyses are designed. Such benchmarks currently only exist for a few research areas and even where they exist, they are often not effectively maintained, due to the required manual effort. This makes evaluating new analyses and tools on software that relies on current technologies often impossible.

We describe ABM, a methodology to semi-automatically mine software repositories to extract up-to-date and representative sets of applications belonging to specific domains. The proposed methodology facilitates the creation of such collections and makes it easier to release updated versions of a benchmark suite. Resulting from an instantiation of the methodology, we present a collection of current real-world Java business web applications. The collection and methodology serve as a starting point for creating current, targeted benchmark suites, and thus helps to better evaluate current program-analysis and software-engineering tools.

Categories and Subject Descriptors C.4 [*Measurement Techniques*]

Keywords benchmark suite, collection, ABM methodology, automated

1. Introduction

It is a challenging task to evaluate new static or dynamic analysis algorithms (e.g. for policy enforcement, code optimization, or finding security issues) and software-engineering tools (e.g. code-recommendation systems or code visualizations). For the sake of reliability, reproduction, and comparison with previous approaches, using well-known, established benchmarks is often considered the best solution to evaluate the quality of a new analysis. Among the more established suites are the DaCapo benchmarks [4], the Scala Benchmarking Project [14], the Qualitas Corpus [16], SecuriBench [12], or micro-benchmark suites such as DroidBench [3], SecuriBench Micro, or PointerBench [15].

New analyses and tools sometimes target domains that have not been researched before. Their target programs may vary, and evaluating those approaches on commonly-recognized benchmarks may not show their full potential. Furthermore, all suites mentioned above except for DroidBench and PointerBench are no longer maintained. This makes it impossible to evaluate new analyses and tools developed for current technologies. This can prompt authors to create their own benchmark suites, like DroidBench for Android applications, SecuriBench for web applications, or PointerBench for pointer analyses. Alternatively, in lack of current, suitable benchmark suites, the authors of scientific papers are frequently forced to evaluate their approaches on software that is chosen in an ad-hoc fashion from popular open-source projects on SourceForge, BitBucket or GitHub, or even self-made applications. This significantly hinders the comparison of analyses and tools.

In this paper, we present the Automated Benchmark Management (ABM) methodology to facilitate the management of up-to-date, reliable, and domain-specific benchmark suites for program-analysis and software-engineering tool evaluation. ABM seeks to maximize the automation of the benchmark creation and maintenance processes. By defining a set of filters, ABM uniformizes the process of benchmark creation for a particular target domain. Automatically reusing those filters creates updates of the benchmark suite, maintaining a continuity between the different versions of the benchmark while keeping it up-to-date.

We also present an instantiation of this methodology for Java business web applications, and the benchmark suite it yields. The suite can be used to study how current web applications are built, or, for instance, to evaluate tools and analyses that attempt to detect vulnerabilities in such applications. Both the instantiation of the methodology and the resulting collection are made available.

The remainder of this paper is organized as follows: Section 2 describes the ABM methodology. Section 3 details the instantiation of the methodology for Java business web applications, and the obtained collection. In Section 4, we discuss the limitations of our methodology, the evaluation plans and the next steps. Section 6 presents existing benchmark suites and related automation approaches.

2. The ABM Methodology

In this section, we detail the ABM methodology by presenting its overall goals, and discussing the process.

2.1 Benchmark Properties

As mentioned by Tempero et al. [16], four aspects should be considered when designing a code corpus: *size*, *content*, *representativeness*, and *permanence*. Existing benchmarks vary in size. DaCapo-9.12-bach, for instance, contains 14 large-scale projects, while DroidBench comprises more than 100 apps, all of which are very small. To ensure code diversity, ABM includes as many suitable projects as possible, regardless of their size.

The contents of a collection created with ABM should help evaluating new tools and analyses. For this purpose, collections should only include publicly available projects. Additionally, because many

[Copyright notice will appear here once 'preprint' option is removed.]

analysis frameworks operate on bytecode, they should provide both: the source code and the binaries, and ensure that the source code is compilable.

The projects included in an ABM collection should be representative of real-world software in the targeted domain.

As an additional requirement, a collection should be up-to-date, i.e. only contain new or still maintained projects at the time of its creation. This property ensures that the collection can be easily updated to reflect current code usage.

To summarize, collections obtained from ABM should be:

- up-to-date,
- representative of the real-world,
- only contain open-source projects,
- only contain compilable projects, and
- provide both the source code and the binary files

2.2 The Methodology

The ABM methodology consists in the choice of a source from which to obtain interesting projects, and a set of filters which ensure that the projects contained in the final collection follow the properties presented in the previous section.

2.2.1 Source

The most fundamental task when creating a benchmark suite is to identify the source from which it is possible to collect a representative collection of projects. Popular repository platforms like GitHub, BitBucket, or SourceForge are widely used and contain projects developed by (commercial) organization as well as individual programmers. Projects on these repositories are very active, and host a high diversity of open-source projects. Such platforms are therefore well suited as starting points for open-source benchmark collections. Due to the breadth and sheer number of the projects found on these repositories, it is even possible to create benchmarks for areas such as enterprise applications which typically do not lend themselves toward open-source. Selecting projects from these repositories can be seen as a best-effort to approximate industrial, real-world software.

The first step of the ABM methodology is to identify which software repositories to mine, and how to extract useful metadata. Some platforms offer query APIs such as the GitHub REST API, which can be used to query projects and meta information. Independent tools such as GHTorrent [10] and GitHubArchive [9] can also be used to mine GitHub.

When selecting projects, a filter should be created to ensure that only active (up-to-date) projects are chosen. This can be done, for example, by checking the date of the last commit.

2.2.2 Representativeness Filters

An important aspect of a benchmark is its representativeness. Projects should be representative with respect to the target domain of the evaluated approach (e.g. Android applications), and to the evaluated tool or analysis (for example a taint analysis would target projects containing certain source/sink API calls, whereas a pointer analysis would be interested in projects containing particular types of assignments).

As the notion of representativeness varies according to what the user wants to evaluate, it is left for them to define, as it is the case for the currently existing benchmarks suites. As opposed to those benchmarks, where the user must choose among the projects available in the suite, ABM directly provides representative projects, tuning the dataset towards the desired domain.

For this step, the properties of the target projects should be defined as closely as possible based on the projects' domain or the tool's domain. Based on this criteria, a set of filters that select relevant projects from the dataset is created. These filters can be based on metadata made available by the repository platforms or on code-specific information obtained by analyzing the project.

2.2.3 Build Filters

Many projects present on repository platforms are simple test projects, libraries, program fragments, or simply random code dumps. While some of them would be suitable small test cases for source code-based analyses, the ABM methodology aims at approximating real-world programs, which is often reflected with the usage of a mature build system. For a project to be included in the collection, the build system should be recognized, and the project should compile and produce an executable. This is ensured with an additional set of filters.

2.2.4 Methodology Workflow

The combination of the filters defined in the previous paragraph applied to the chosen repository platforms results in an instantiation of the ABM methodology that, when applied, yields a collection that follows the properties defined in Section 2.1. The filters can be applied in any order, as shown in the instantiation of ABM for Java business web applications in Section 3.

Once the collection is created, it can easily be kept up-to-date, by re-running the instantiation. A collection's maintenance can be fully-automated, the manual overhead of the methodology being the initial definition of the filters and source repositories.

We will discuss the limitations to this semi-automated approach in Section 4.

3. ABM for Java Business Web Applications

We have applied ABM to the case of Java business web applications, yielding the ABM instantiation shown in Figure 1. By applying it, we obtained a collection of Java business web applications.

Java is of particular interest as it is used by 26% of the top 1000 most visited sites [18], and is the target of a lot of research. Furthermore, no up-to-date, established benchmarks exist for this particular case.

3.1 Instantiating ABM

3.1.1 Source

We choose GitHub as our sole project source due to its high popularity in the targeted domain. It contains many business web applications, including recent student projects, open-source collaborative applications, and also open-source commercial projects.

To query for projects and retrieve metadata, we use the GitHub REST API, which allows easy access to the repositories and their metadata. Using intermediate querying platforms is also possible, but would make little difference in our use case.

An *active projects* filter is created using the date of the latest commit from the metadata. Half of the projects were not updated in the last year. They are considered unmaintained and are removed from the dataset.

3.1.2 Representativeness Filters

To initialize the methodology, we identify the key properties of the projects that should be included in the collection. In this case, we only query for projects written in Java as we want to create a Java business web applications benchmark. In the same query, we also specifically ask for the three main types of business web applications: customer relationship management (CRM), content management system (CMS), and enterprise resource planning (ERP) [17]. GitHub returns a total of 2116 projects.

GitHub sometimes mistakenly tags projects with the wrong programming languages, or the initial search query sometimes

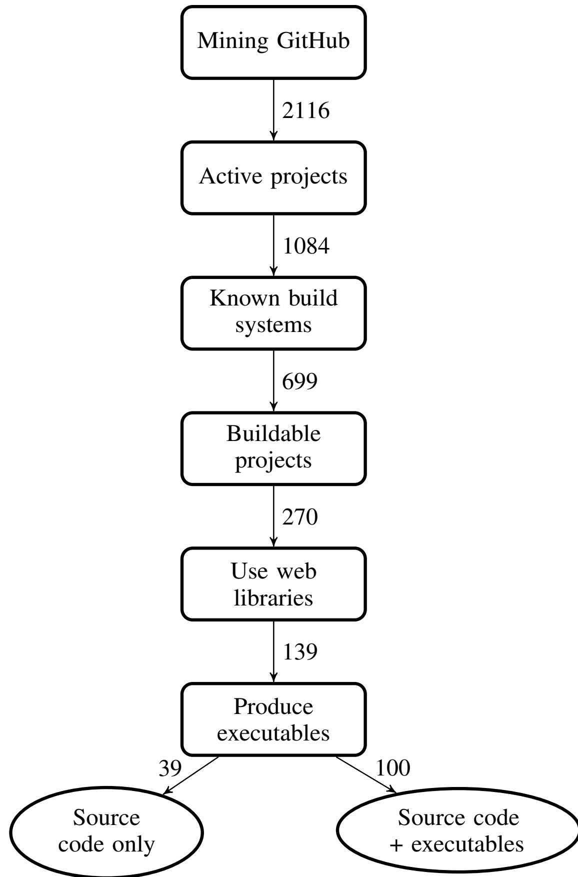

Figure 1. An instantiation of ABM for Java business web applications, with the number of remaining project after each phase, and the location of the different filters (circled).

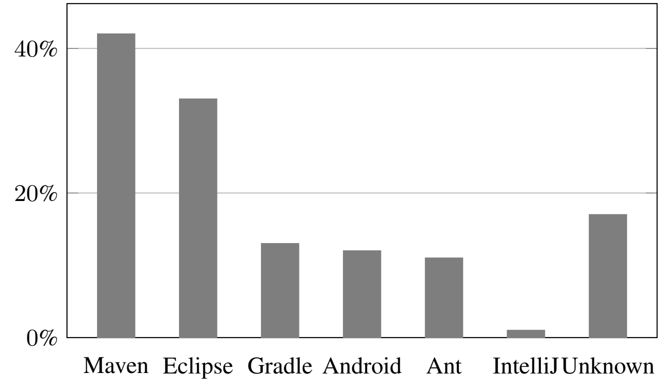

Figure 2. Build systems distribution in the dataset. One project can have several build systems.

returns projects that are out-of-scope. We found that the use of libraries is a sensitive metric to further filter the dataset, as it not only reflects the programming language of at least part of the application, but it also shows which APIs are used by the application, allowing us to infer the application's functionalities. For example, an application using the javax servlet API is most likely a Java web application. Thus, we introduce a *web libraries filter* that extracts project libraries declared in the applications' build files, or found as `.jar` files in the projects. Those libraries are then compared to a whitelist created by a domain expert. Overall, half of the buildable projects do not contain a whitelisted library, and are removed from the collection.

3.1.3 Build Filters

We consider IDE-built projects as not relevant for the collection, as in such cases the builds are often not repeatable across different environments. A *known build systems filter* selects projects that use known build systems. Such projects are typically more mature, useful and representative of real projects. As shown in Figure 2, the most popular build system for Java web applications in the dataset is Maven. We only include projects that can be built using Ant, Maven, or Gradle, thus removing a third of the remaining projects. Those include all projects that are out of scope (Android applications), do not have a proper build system (Eclipse and IntelliJ), or have a build system which could not be clearly identified (Unknown). The latter case can be due to missing or custom build files, projects not written in Java, or that are not web applications.

The second *build filter* attempts to build the projects to ensure that only compilable projects are kept. We use the standard build commands: `gradle build`, `mvn compile`, and `ant build` or `ant compile`. Around 40% of the projects compile well. The others either have incorrect code, or require specific configurations that our test environment does not provide.

Executables are created using the standard build commands: `gradle jar/war/ear`, `ant jar/war/ear`, and `mvn install`. Some of the projects do not produce an executable, due to custom or missing configurations, or environment issues. Those projects are kept in the collection, to provide more material for source-code analyses, but no executables are provided.

3.1.4 Methodology Workflow

Figure 1 illustrates the instantiation of the ABM methodology for Java business web applications. All filters before the "Known build systems" step (incl.) are computed based on project metadata. After this step, we download the remaining projects and apply the next steps on the local copies. The filters are ordered specifically to minimize the number of queries made to the GitHub API and the number of downloaded projects. Once the instantiation is complete, it is fully automated and can then be run regularly to keep the collection up-to-date.

Approximately 6.5% of the projects returned by the initial query remain after the filters are applied.

3.2 Resulting Collection

The resulting collection contains the source code of 139 projects. Deployable binaries (`.war`, `.ear` or `.jar` format) are provided for 100 of them. The projects vary in size (5 to 379.002 Loc), activity (1 to 323 commits in the last year) and contributors (1 to 15). They range from simple test projects used to learn business web application development, to large-scale real-world web applications like Enonic [8] and AppDynamics [2] which are CMS and CRM solutions deployed in industry.

The collection as well as the scripts and filters used to create it are available at: `http://www.st.informatik.tu-darmstadt.de/artifacts/webapps/`.

Both the source code and the binaries are made available. An index file in `.xls` format lists those resources. It contains metadata about the different projects, namely build and library information, and a link to the original GitHub repository and chosen commit. The `.xls` file also contains the whitelist used for the libraries filter.

4. Limitations

The presented ABM methodology is subject to three main limitations. A first limitation is that the collections created with ABM are only limited to open-source projects, since examples of applications that are developed by companies for in-house purposes are rarely made publicly available. Nevertheless, ABM also retrieves projects that are developed for commercial purposes, and it is reasonable to assume that these projects are developed using the same, or even better standards than typical in-house projects.

Second, to keep the build process fully automated, we only consider projects that have well-defined build scripts. This filters out some potentially interesting projects, such as the commercial project Adempiere [1]. These projects have very stringent requirements on the build environment and/or have very complex build dependencies that cannot easily be extracted. Given that the collection contains some very large projects (≈ 300KLoc), we do not consider this a severe limitation, but we are still considering to improve our scripts in this regard.

The third limitation is that the methodology relies on the user-defined notion of representativeness, which can be difficult to translate into sensible filters. For example, the ABM collection we retrieved comprises CMS, CRM and ERP applications. However, GitHub returned other applications such as INCF's EEG/ERP database for electroencephalography experiments [11]. Since these projects also contain Java modules and have well defined build scripts, they are also included in the ABM collection, despite being out-of-scope w.r.t. to selected domain. As a result, ABM generally yields a collection of applications that is a significant step toward a benchmark suite, but some additional (potentially manual) filtering may be required.

Sharing a filter set effectively allows different researchers to produce collections for the same target programs and analyses. The collections might differ if they were created at different times. Because two researchers would most likely define different filters for the same targets, we advocate the use of a collaborative platform for ABM based collections.

5. Using the Benchmark Collection

We have used the proposed collection to evaluate static taint analyses and novel algorithms for dead code elimination [7]. For the dead code elimination, we were particularly interested in learning how useful the analysis is for web applications, compared to analyzing the Java Runtime Environment, or the Qualitas Corpus. The analysis found far fewer dead paths in the code of the web applications than it did for projects from the Qualitas Corpus, or the JDK. For the JDK, it found approximately one issue every 20 methods. In the case of the largest web application from the ABM collection, one issue was reported every 312 methods. For other web applications, the ratio was even smaller. This result is expected, as age seems to be a major reason for dead code, and web applications – from the ABM benchmark in particular – are generally (much) younger. Those preliminary results show that the choice of benchmarks matters a lot, and that methodologies such as the one proposed may help in improving the evaluation conditions of analysis tools.

With our experiments for taint analysis, we hope to better assess the usefulness of our static taint-analysis tools which were developed for Android, in the context of Java web applications.

6. Related Work

The DaCapo benchmark suite [4] is comprised of open-source, real-world projects containing important memory loads. The suite is mainly used for performance testing. Its latest release is the DaCapo-9.12-bach suite, released in 2009.

In their paper, Tempero et al. discuss the challenges of creating and maintaining a corpus of representative programs. Their reasoning produced the Qualitas Corpus [16], a large collection of open-source Java programs. The collection's latest distribution was released in 2013.

Created in 2012, the Scala Benchmarking Project addresses the problem of the absence of benchmarks for the use of non-Java languages on the JVM. It contains real-world Scala applications that can be used to compare performance of Java and Scala programs.

SecuriBench [12] was created to respond to the lack of Java web application benchmarks to assess static analyses by Livshits et al. It contains large, real-life applications for J2EE programs, and was last updated in 2005. SecuriBench Micro also targets Java web applications, and is made of smaller, hand-crafted projects containing known vulnerabilities. It was last updated in 2006.

Created to assess the precision of the taint analysis FlowDroid [3], DroidBench is a micro-benchmark comprised of many custom-made, small Android applications, specifically targeting the weaknesses of data-flow analyses for Android.

PointerBench [15] is a micro-benchmark created for the pointer analysis Boomerang in 2016. Due to the lack of existing benchmarks for pointer analyses, Späth et al. created PointerBench to compare the precision of their approach to existing pointer analyses.

These collections and benchmark suites were all created to address the lack of existing benchmarks in the research areas needed by their authors. The projects they contain were either meticulously hand-picked from existing real-world applications, or hand-made to target specific vulnerabilities and weaknesses in analysis tools. For this reason, many of those collections became reference benchmark suites in their respective domains, and are widely used to compare and test new approaches. However, the maintenance of such benchmarks is a time-consuming, and a somewhat repetitive task. They are no longer maintained, to the exception of DroidBench and PointerBench, which is a recent creation. Our approach thrives to automate the collection process to the largest extent, facilitating the creation and automating the maintenance of benchmark suites.

Efforts have been made toward automatizing the process of benchmark creation. In their paper, Dallmeier et al. present iBugs [5], an approach that automatically analyzes the history of real-world projects to extract bugs, from which it derives minimum test cases. Its latest update was made in 2011.

JSBench [13] was created by Richard et al. as an approach to automatically create micro-benchmarks for JavaScript performance and load issues. It records human interaction with a website that uses JavaScript, records traces containing heavy workloads, and arranges them make them deterministic. The benchmark suite was last updated in 2013.

In his paper [6], Dujmović discusses techniques to create fully artificial benchmarks on-demand, based on simple criteria such as program size, or basic operations characteristics.

Such approaches thrive to automate the process of benchmark creation. The test cases generated from the first two approaches are representative of real-world issues to a certain extent, as they are extracted from real code, and observed problems. These approaches can be used to create micro-benchmark suites, and their test cases are either modified (eg. for the sake of determinism), or completely artificial. In contrast, the ABM methodology aims at collecting unchanged real-world applications to avoid the limitations related to the use of micro-benchmarks. The first two approaches are especially interesting in how they determine relevant test cases, with respect to the target analysis. In future work, we plan to base ourselves on such criteria to characterize representativeness towards the evaluated analysis.

7. Conclusion

In this paper, we discussed how to create more dynamic, easily updatable benchmark suites that can be adapted to the needs of different research topics. We have presented the ABM methodology for semi-automatically creating and automatically maintaining collections of real-world open-source applications within a particular domain. Resulting from an instantiation of this methodology, we have created, released, and discussed the ABM collection, a suite of 139 projects within the domain of Java web applications. Both the methodology and the collection serve as a starting point for creating current, targeted benchmark suites with the goal of better evaluating program-analysis and software-engineering tools. We have also discussed the limitations of the ABM methodology, such as the necessity to manually filter the selection of benchmark application for true representativeness. In future research, we plan to determine how to better characterize representativeness with respect to real-world applications and the evaluated program-analysis or software-engineering tool. One possibility to further automate the creation of filters would be to apply machine-learning to an initial set of projects to extract interesting features – such as library information for example, and use them to discover similar projects.

Acknowledgments

This research was supported by a Fraunhofer Attract grant as well as the Heinz Nixdorf Foundation.

References

[1] Adempiere. Adempiere home page. `http://adempiere.org/site/`.

[2] AppDynamics. Appdynamics home page. `https://www.appdynamics.com/`.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 29, 2014.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *21st Conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM, 2006.

[5] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 433–436, New York, NY, USA, 2007. ACM.

[6] J. Dujmović. Automatic generation of benchmark and test workloads. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, WOSP/SIPEW '10, pages 263–274, New York, NY, USA, 2010. ACM.

[7] M. Eichberg, B. Hermann, M. Mezini, and L. Glanz. Hidden truths in dead software paths. In *10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015. ACM, 2015.

[8] Enonic. Enonic home page. `https://enonic.com/`.

[9] GithubArchive. Githubarchive home page. `https://www.githubarchive.org/`.

[10] G. Gousios and D. Spinellis. Ghtorrent: Github's data from a firehose. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 12–21, June 2012.

[11] INCF. Incf home page. `http://www.incf.org/`.

[12] B. Livshits. Defining a set of common benchmarks for web application security. In *Proceedings of the Workshop on Defining the State of the Art in Software Security Tools*, Aug. 2005.

[13] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of javascript benchmarks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 677–694, New York, NY, USA, 2011. ACM.

[14] A. Sewe. *Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine*. PhD thesis, TU Darmstadt, Darmstadt, Okt. 2012.

[15] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In *ECOOP*, 2016. To appear.

[16] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Asia Pacific Software Engineering Conference*. IEEE, 2010.

[17] Web Technology Surveys. Usage of content management systems for websites. `http://w3techs.com/technologies/overview/content_management/all`.

[18] Web Technology Surveys. Usage of server-side programming languages broken down by ranking. `http://w3techs.com/technologies/cross/programming_language/ranking`.