On Limitations of Modern Static Analysis Tools

Andrew Walker, Michael Coffey, Pavel Tisnovsky and Tomas Cerny

Abstract Static analysis is one of the most important tools for developers in the modern software industry. However, due to limitations by current tools, many developers opt out of using static analysis in their development process. Some of these limitations include the lack of a concise, coherent overview, missing support for multiple repository applications and multiple languages and lastly a lack of standardized integration mechanisms for third-party frameworks. We propose an evaluation metric for static analysis tools and offer a comparison of many common static analysis tools. To demonstrate the goal of our metric we introduce the Fabric8-Analytics Quality Assurance Tool as a benchmark of a tool which successfully passes our evaluation metric. We demonstrate usage of this tool via a case study on the Fabric8-Analytics Framework, a framework for finding vulnerabilities in application dependencies. We issue a challenge to developers of modern static analysis tools to make their tools more usable and appealing to developers.

Keywords Static · Analysis · Multi-repository · Automation

1 Introduction

Static analysis is the process of automatically analyzing source code using methods such as rule checking, lexical analysis and pattern matching [1]. It should be an important part of the developmental process as performing static analysis on a given project can find issues, potential vulnerabilities and improve quality assurance (QA) [2]. However, static analysis is an umbrella term that can be split up into a multitude of diverse processes. Most of these processes are run on different tools, so the developer

A. Walker (✉) · M. Coffey · T. Cerny (✉)
Computer Science, Baylor University, Waco, TX 76798, USA
e-mail: Andrew_Walker2@baylor.edu

M. Coffey
e-mail: Michael_Coffey@baylor.edu

P. Tisnovsky
Red Hat Czech, FBC-Purkyova 99, Brno 612 00, Czechia

K. J. Kim and H.-Y. Kim (eds.), *Information Science and Applications*,
Lecture Notes in Electrical Engineering 621,
https://doi.org/10.1007/978-981-15-1465-4_57

needs to run an array of tools to show enough results for the static analysis to be worth the time. Because of this, a large amount of developers does not believe the static analysis process to be worth the time and effort it takes to find and run all the tools needed.

In order for the static analysis process to be appealing to developers it needs to address three main issues we've identified with current static analysis methods. Current static analysis tools are well suited to one specific task; however, a large repository may use many tools that will generate a large volume of output. A static analysis tool needs a way to (1) show results in a concise, comprehensive way. This way the developer can look at one screen and determine the results from multiple static analysis tests. Furthermore, many static analysis tools only perform the static analysis on a given file or repository. Static analysis tools must offer (2) native multiple repository support. By extension to the previous issue, static analysis tools must be able to (3) integrate into third-party dashboards and frameworks. Currently, no tools offer standardized integration support for frameworks. Lastly a static analysis tool should (4) support multiple languages as no application is usually written in a single language.

In this paper we show that the current approach by static analysis tools is insufficient for many use cases faced by modern developers. The problems of integration, multi-repository support and a centralized view remain unsolved by the vast majority of static analysis tools. We compare multiple static analysis tools and define an evaluation framework for existing features and advantages for readers to consider. We present a solution to the problems discussed above by applying our evaluation framework in analyzing a case study on one tool we identified to support most of the considered features.

The rest of the paper is outlined as follows. Section 2 provides background information and compares existing static analysis tools, Sect. 3 is a case study considering a tool we identified to be most promising, Sect. 4 presents the conclusion and future work.

2 Related Work

It is important to catch errors in code early in the development process as the cost of fixing the errors drastically increases the further along the bug is [3]. Specifically, the costs involved with a vulnerability found after deployment costs four to eight times more than when they are found and dealt with prior to deployment [4]. The National Institute of Standards and Technology estimated that around $60 billion dollars a year is wasted due to faults in software [4]. Clearly errors in the code should be taken seriously and dealt with swiftly. Studies on analysis techniques [5], noted that using static code analysis to discover violations can reduce production costs up to potentially even 23%. Static code analysis is also beneficial in that the same research also showed an increase of discovered violations over previous methods by 2.6 times [5].

One way that developers seek to find violations in code is to enforce good programming practice through the use of tools that use a standard coding style [6–10] to analyze their source code for deviations from the accepted standard. These tools vary in their implementation [11, 12] and their method of analysis so deciding which tool to use to analyze a repository is equivalent to picking the standard to adhere to.

One such tool, CheckStyle [7], written for Java, focuses on design level errors rather than finding errors in the details. It can find problems within the class and method design as well as checking the layout and formatting of the code. While this tool is good for a higher-level analysis, design decisions oftentimes propagate throughout the entire project, not just a single class or method. CheckStyle is limited to analysis of a single file at a time. To analyze an entire application or integrate into existing continuous integration pipelines would require significant overhead from the developer. This vastly limits the usefulness of this tool for development. Like many of the tools discussed in this paper, this tool is not packaged with a graphical user-interface (GUI) which makes interpretation more difficult for the developer. Furthermore, this tool requires significant configuration and explicit definition of what errors to check for. If a developer misconfigured this tool, errors would persist without any warning.

FindBugs [6, 13, 14] is a static analysis tool written for Java which aims to fix some of the issues present in the CheckStyle tool. For starters, it uses bug pattern matching with over 60+ patterns to find many common occurring bugs [14, 15] which means FindBugs does not require out-of-the-box configuration, but rather is already configured for three categories of code violations. These violation categories are correctness bug, bad practice and lastly dodgy. Each of these violation categories offer different suggestions on how to handle the violation. A correctness bug is likely a programming error that should be addressed, a bad practice violation is a deviation from standard coding practice and could be addressed and dodgy violations are code that is merely confusing. The grouping of issues helps address the issue of a lack of centralized view in CheckStyle and FindBugs also comes with a GUI and can be run as an application, instead of via the command line. One of the most important improvements of FindBugs is that it is not limited to a single file like CheckStyle, however FindBugs does not offer any native support for multi-repository applications. Lastly, FindBugs does offer command line utilities and better continuous integration support but lacks a standardized API for integration with third-party frameworks. Another drawback of this tool is that it only supports Java and is no longer supported. A spiritual successor SpotBugs [10] has continued support and added plugin support which makes it more usable in the development process. The analysis of Spot-Bugs comes in the form of a list of errors and warnings, usually in an integrated development environment (IDE) window which still lacks the quick overview that is often desired for a repository. It is not always necessary to know every detailed error but rather a holistic overview of the code. Lastly, like FindBugs, SpotBugs is limited to only Java code analysis.

While Java remains one of the most popular languages today, scripting languages like Python are consistently becoming more apparent in the development of large-scale project. Python adds its own set of issues in static analysis due to the lack of

types and the more restrictive formatting rules. Pyflakes [8] is a Python only static analysis tool which aims to provide a thorough analysis of Python applications. This tool works by parsing the source code file into an Abstract Syntax Tree (AST). It works file by file, building the tree and then analyzing it. Errors are found by searching through the nodes of the AST for certain patterns that denote errors. Due to the nature of only checking the AST, this tool is limited in finding errors that would be present in an AST, e.g. unused imports, unused variables. Additionally, this tool does not check the style of the Python code. Overall this tool lacks many features present in tools for other languages and does not offer any support for multiple repositories or framework integration.

Pylint [9] is an attempt at solving the issues present in Pyflakes by offering a more thorough and robust static analysis tool. It fills in the static analysis gaps present in Pyflakes by offering code style checking as well as more exhaustive error checking. Pylint also provides UML diagram generation and extensibility features for creating plugins for an IDE. In an effort to offer more complete integration support, Pylint offers full continuous integration support however like the other tools mentioned, lacks a standard API for third-party framework support. Pylint still lacks support for any language other than its native Python. PMD [16] is a static analysis tool which is different than the previously mentioned tools in that it natively supports multiple languages. The tool is configured for rule-based code analysis and also comes packaged with Copy-Paste Detector (CPD) [17] which is an additional library for finding Type-1 [18] code duplication. PMD also offers continuous integration support and an API for developers to extend PMD's rule-set, however not an API for integration into third-party frameworks.

We have identified one tool which offers a variety of features. The Fabric8-Analytics (F8A) [19] Quality Assurance Tool runs multiple static analysis tools and displays the results aggregated into a centralized web portal. This tool combines other static analysis tools to perform broad static analysis. It can perform the analysis on multiple repositories in a given project. The tool's architecture support extension into various languages with the primary tool being Python.

Table 1 Comparison of static analysis tools

Tool	Centralized overview	Framework integration	Multi-repository	Multiple language support
CheckStyle				
Pyflakes				
Pylint	X			
SpotBugs	X	X		
FindBugs	X	X		
PMD				X
Fabric8-analytics	X	X	X	X

In Table 1 we summarize considered tools and compare them whether they support the features such as centralized overview from various features, enable integration with other tools, supports multi-repository evaluation and could be extended for various programming languages.

3 Case Study

While the tools discussed above provide varying levels of code analysis and introspection, we identified only one tool that natively supports multi-repository analysis and has support for multiple languages. The majority of tools either focus on specific aspects of code, like type checking or coding style and lack support for many other important aspects that need to be checked during development or the tools lean towards holistic continuous integration support and are difficult to interpret and use on a large scale project. A lot of development effort goes towards application artifacts like documentation and an analysis tool for an application should cover analysis of these artifacts as well. Furthermore, an application is not usually limited to one specific language. Any analysis tool for use with an application should be able to handle multiple languages with minimal configuration. Lastly, a detailed report which lists the errors in an application is not easy to manage or interpret by a developer when an application covers many repositories, especially when those repositories grow large. A static analysis tool should provide an easy way to grade the aspects of the application and provide a quick overview for developers. Below we will introduce Fabric8-Analytics static analysis tool and walk through a use case for the tool.

3.1 Introducing Fabric8-Analytics

One tool we have found to address the evaluation topics from Table 1 is the Fabric8-Analytics QA Tool (F8A QA). It has been developed for managing Fabric8-Analytics Framework (F8AF), a framework meant for vulnerability checking on dependencies for Openshift.io languages (Java, Javascript, Python). This framework also checks the licenses of application's dependencies against your application's license for discrepancies. Due to the large number of repositories needed for this framework, an additional product was produced in order to manage quality assurance of the framework. This F8A QA shows the code coverage, documentation results, linter results, dead code, common issues, cyclomatic complexity, maintainability index, and overall status for all the repositories in a given project. Below we briefly discuss the F8AF, however, the focus of our case study is on F8A QA.

The F8AF is a framework to detect vulnerabilities in repositories and to detect vulnerabilities in the dependencies of an application. The problem with most vulnerability dependency checkers is that they will only run for one language. F8AF was developed for a number of languages to specifically target this problem within

existing dependency analyzers. The framework was developed as an extension to Openshift.io, a wrapper that Red Hat developed to encompass Kubernetes, making deployment of projects more straightforward for the developers. The primary purpose of F8AF was to run dependency analysis on applications hosted within Openshift.io. In addition to the Openshift.io support, F8AF is a plugin that can be downloaded in Visual Studio Code (VS Code). The plugin reports dependencies that have a high Common Vulnerabilities and Exposures (CVE) score, reports unknown or restrictive licenses in dependencies, suggests dependencies to add and replace in the application and shows the dependencies that have been analyzed based on popularity and version.

While developing the F8AF the problems outlined previously with performing static analysis for quality assurance quickly became apparent. The framework spans 30 repositories with each repository containing upwards of 170 source code files. While there are tools that exist to look at files individually or a project individually to run static analysis, there were no tools that could run static analysis or provide a quick overview of an application with that many repositories. In order to solve this problem, F8A QA was developed in parallel to the framework. This tool was developed as a way of providing a centralized view on a multi-repository application running multiple static analysis tools. Additionally, F8A QA was released as a completely open-source project[1] to support integration into other third-party frameworks. In this case study we examine the results from usage of this tool on the F8AF.

3.2 Performing Common Tasks, the Case for a Framework

One of the problems that developers have with static analysis tools is that there are so many of them and they each perform their own separate form of static analysis. As mentioned in the related work section of this paper, static analysis tools are usually experts on one type of static analysis but are not designed to do other types. This is why there is a need for a static analysis framework that brings together an array of static analysis tools in order to produce a single tool for the developer to run over their source code. In order for modern developers to perform static analysis on their software they are forced to find the multiple static analysis tools that best fit the source code that they are writing. Once these tools are found the developer then must run their source code through each tool individually and use the results of each tool to piece together a diagnostic of the source code. This is not an appetizing process, because it takes time away from the developer that they could have had dedicated to other acts in the development process.

F8A QA uses multiple static analysis tools to give developers all the statistics needed for a productive development. It performs static analysis on a given project by running linter testing, docstyle testing, code coverage, dead code checking, common issue checking, cyclomatic complexity testing, and maintainability testing.

[1] https://github.com/fabric8-analytics/fabric8-analytics-common.

– *Docstyle* is a static analysis statistic used for checking the repositories compliance to the language's documentation conventions.
– *Linter* is the process of using static analysis to determine what code could be prone to errors by checking the style of the code.
– *Code coverage* is the term used to describe the amount of code covered by unit tests. An ideal repository would have 100% code coverage from unit tests which would mean that the tests fully cover the code. Code coverage reports can be generated in a number of ways but analyzing a large number of reports is quite tedious for the developer.
– *Cyclomatic complexity* (CYC) is the count of the number of decisions in the source code. The complexity of the source code increases as more decisions are made in it. It is typically good practice to have a lower CYC number because simple source code is easier to maintain than complex source code.
– A *maintainability index* is used to show how maintainable, meaning how easily it is to support and change, a system is. There are many different ways to calculate this index, because maintainability is a vague term. In correlation to this, there are many different tools used to calculate a maintainability index, the F8A QA Tool currently uses Radon [20] for this purpose. Radon uses a complex algorithm made up of the cyclomatic complexity, Halstead volume, and the number of lines of source code in its calculation of maintainability [21, 22].

By black-boxing the specific results from the user, F8A QA supports multiple languages since the actual implementation does not matter in the statistic presented to the user. For example, a repository with half Python and half Java source code files could be analyzed in a cohesive way by F8A QA since the Java could be analyzed with Java specific tools and the Python with Python specific tools with results that are then aggregated into a single statistic for the user to process.

The results of all of these tests simplified are presented in a color-coded, tabular format. An example of the dashboard for F8A QA can be seen in Fig. 1. This process

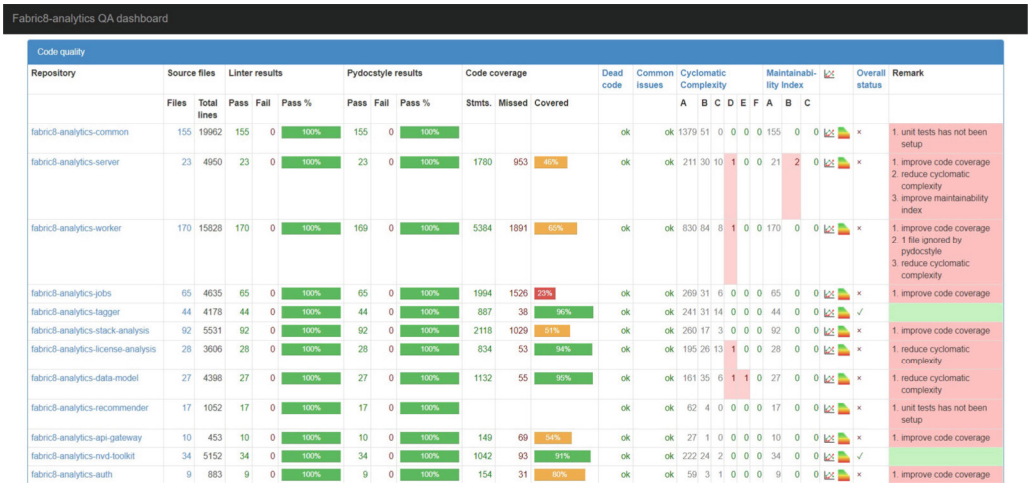

Fig. 1 Fabric8-analytics QA tool dashboard

of simplifying results and grading specific files brings transforms the results of static analysis tools into a simple to understand statistic that presents an easily comprehended overview of the repository. The Linter results, Pydoc-style results, and code coverage sections are shown as breakdowns of number of files that pass, number of files that fail and as a percentage of files that pass their respective tests. Dead code and common issues sections show how many files failed their tests. The cyclomatic complexity and maintainability index sections are shown as grade scales with the determined number of files placed under the grade given to them. CYC grades run from A to F and maintainability index grades run from A to C. Lastly, each repository is given an overall grade from F to A+++ based on the previous statistics in addition to remarks to the developer on how to improve the grade of the repository.

3.3 Sample Use Case

The use case we consider for F8A QA is the F8AF. The F8AF spans a total of 30 repositories which each must have static analysis however most static analysis tools have no way of running on multiple repositories. For these static analysis tools to be used on a multi-repository project the developer would need to run the tool on every repository individually. This process would take an exhaustive amount of time that the developer could have been using more efficiently. The F8A QA offers native support for static analysis of a multi-repository system by running the tools on each repository in the configuration. It is able to run this static analysis with relatively fast speeds. For the 30 repositories in the F8AF, it only took 78 s to run all the static analysis tools on every repository. This was achieved with an Intel i8 Quad Core 2.8 GHz and 8 GB of RAM. The results for some of the repositories of F8AF are visible in Fig. 1.

Usage of F8A QA on the F8AF drastically increased the overall quality of the application. Since applying the tool, documentation coverage has increased to an average of 98% and passing linter results have increased to 99% across the repositories. The results from the dead code and common issues testing is where the usage of F8A QA really shines. Figure 2 is a graph showing the total number of common issues over time in fabric8-analytics-common, the largest repository in F8AF with 155 files and almost 20,000 LoC. Since usage of the F8A QA began, it took around a month to fully identify and fix the common issues in the repository. Even as the LoC steadily increased over the past two years, using F8A QA allowed developers to maintain the high passing percentages in their repository. Results are similar for dead code and these results can be seen similarly in almost all repositories. Currently all but one repository has fully passing dead code and common issues remarks. Code coverage is one of the hardest statistics to increase but with the tool, repositories have seen a steady increase in coverage, with some repositories hitting 100% code coverage. The use of F8A QA has clear, quantifiable benefits in the modern development process and should serve as an example of what static analysis tools should strive for.

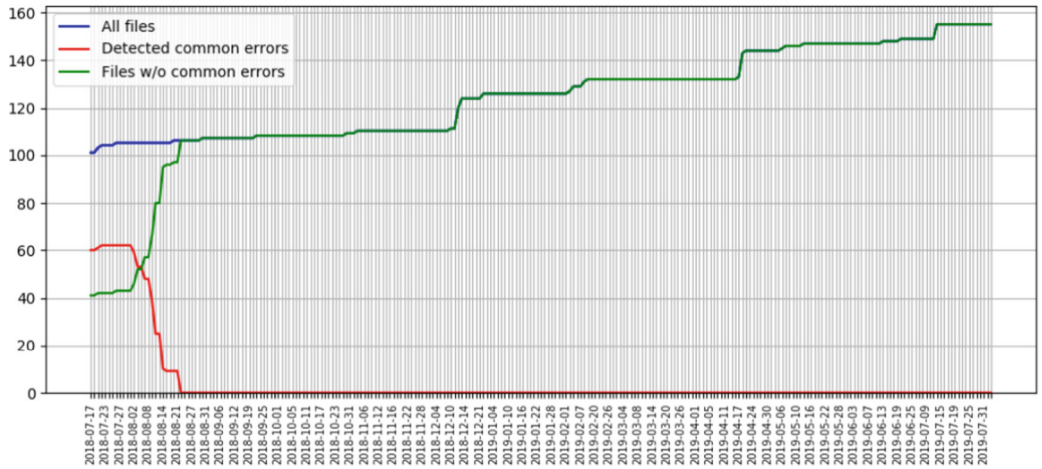

Fig. 2 Common issues in fabic8-analytics-common

4 Conclusion

We highlight the insufficiency, gaps and propose future direction for future static analysis tools that are needed to assist with modern industry development especially relevant to distributed systems and module-based applications. Most common static analysis tools lack a centralized overview, standard framework integration, multi-repository support and multiple language support. However, we identified one tool which proposes solutions to those problems. F8A QA is a static analysis framework with demonstrated validity via the F8AF testbed. F8A QA was able to run on 30 repositories with upwards of 170 source code files in about 78 s. By displaying all these static analysis statistics in a manageable dashboard, it is easy for the developer to extract a quick overview of a multi-repository application. Though the current dashboard implementation is focused on Python, support exists in the modules for other languages.

It is time for static analysis tools to adapt to the new methods modern developers use in the field. Through this paper we are challenging the developers of these static analysis tools to make their tools more appetizing to the developers who need them most. As of this date F8A QA is the only open-source static analysis software of its kind.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. 1854049.

References

1. Li P, Cui B (2010) A comparative study on software vulnerability static analysis techniques and tools. In: 2010 IEEE international conference on information theory and information security, pp 521–524, Dec 2010. https://doi.org/10.1109/ICITIS.2010.5689543

2. Zheng J, Williams L, Nagappan N, Snipes W, Hudepohl JP, Vouk MA (2006) On the value of static analysis for fault detection in software. IEEE Trans Software Eng 32(4):240–253. https://doi.org/10.1109/TSE.2006.38
3. Baca D, Carlsson B, Lundberg L (2008) Evaluating the cost reduction of static code analysis for software security. In: Proceedings of the third ACM SIGPLAN workshop on programming languages and analysis for security, pp 79–88. PLAS '08, ACM, New York, NY, USA. https://doi.org/10.1145/1375696.1375707
4. Telang R, Wattal S (2007) An empirical analysis of the impact of software vulnerability announcements on firm stock price. IEEE Trans Softw Eng 33(8):544–557. https://doi.org/10.1109/TSE.2007.70712
5. Bardas AG (2010) Static code analysis. Rom Econ Bus Rev 4(2):99–107. https://ideas.repec.org/a/rau/journl/v4y2010i2p99107.html
6. Findbugs (2015) http://findbugs.sourceforge.net/
7. Checkstyle (2019) https://checkstyle.sourceforge.io/
8. Pyflakes (2019) https://pypi.org/project/pyflakes/
9. Pylint (2019) https://www.pylint.org/
10. Spotbugs (2019) https://spotbugs.github.io/
11. Manzoor N, Munir H, Moayyed M (2012) Comparison of static analysis tools for finding concurrency bugs. In: 2012 IEEE 23rd international symposium on software reliability engineering workshops, pp 129–133, Nov 2012. https://doi.org/10.1109/ISSREW.2012.28
12. Rutar N, Almazan CB, Foster JS (2004) A comparison of bug finding tools for java. In: 15th International symposium on software reliability engineering, pp 245–256, Nov 2004. https://doi.org/10.1109/ISSRE.2004.1
13. Ayewah N, Pugh W, Morgenthaler JD, Penix J, Zhou Y (2007) Using FindBugs on production software. In: Companion to the 22nd ACM SIGPLAN conference on object-oriented programming systems and applications companion, pp 805–806. OOPSLA'07. ACM, New York, NY, USA. https://doi.org/10.1145/1297846.1297897
14. Vetro A, Morisio M, Torchiano M (2011) An empirical validation of FindBugs issues related to defects. In: 15th Annual conference on evaluation assessment in software engineering (EASE 2011), pp 144–153, Apr 2011. https://doi.org/10.1049/ic.2011.0018
15. Ayewah N, Pugh W, Hovemeyer D, Morgenthaler JD, Penix J (2008) Using static analysis to find bugs. IEEE Softw 25(5):22–29. https://doi.org/10.1109/MS.2008.130
16. Pmd: An extensible cross-language static code analyzer (2019). https://pmd.github.io/
17. Copy-paste detector (2019) https://pmd.sourceforge.io/pmd-4.2.5/cpd.html
18. Sheneamer A, Kalita J (2016) A survey of software clone detection techniques. Int J Comput Appl 137(10):1–21
19. Fabric8-analytics (2019) http://fabric8.io/faq/
20. Radon (2019) https://radon.readthedocs.io/en/latest/intro.html
21. Coleman D, Oman P, Ash D, Lowther B (1994) Using metrics to evaluate software system maintainability. Computer 27(08):44–49. https://doi.org/10.1109/2.303623
22. Oman P, Hagemeister J (1992) Metrics for assessing a software system's maintainability. In: Proceedings conference on software maintenance, pp 337–344, November 1992. https://doi.org/10.1109/ICSM.1992.242525