An Automated System for Testing Static Source Code Analysis Tools

Damir Gimatdinov
Faculty of Computer Science
Higher School of Economics
11 Porkrovsky Boulevard, Building S
109028, Moscow, Russian Federation
damir.gimatdinov@huawei.com

Alexander Gerasimov
Chong-Ming Laboratory
Huawei Russian Research Institute
Derbenevskaya naberezhnaya 7b9
115114, Moscow, Russian Federation
gerasimov.alexander@huawei.com

Petr Privalov
Chong-Ming Laboratory
Huawei Russian Research Institute
Derbenevskaya naberezhnaya 7b9
115114, Moscow, Russian Federation
petr.privalov@huawei.com

Veronica Butkevich
Chong-Ming Laboratory
Huawei Russian Research Institute
Derbenevskaya naberezhnaya 7b9
115114, Moscow, Russian Federation
butkevich.veronika.nikolaevna@huawei.com

Natalya Chernova
Chong-Ming Laboratory
Huawei Russian Research Institute
Derbenevskaya naberezhnaya 7b9
115114, Moscow, Russian Federation
chernova.natalya@huawei.com

Anna Gorelova
Chong-Ming Laboratory
Huawei Russian Research Institute
Derbenevskaya naberezhnaya 7b9
115114, Moscow, Russian Federation
anna.gorelova@huawei.com

***Abstract*—Automated testing frameworks are widely used for assuring quality of modern software in secure software development lifecycle. Sometimes it is needed to assure quality of specific software and, hence specific approach should be applied. In this paper we present an approach and implementation details of automated testing framework suitable for acceptance testing of static source code analysis tools. The presented framework is used for continuous testing of static source code analyzers for C, C++ and Python programs.**

Keywords—automated testing, quality assurance, static source code analysis.

I. Introduction

Acceptance testing is a very common approach to make sure required software functionality is satisfying needs of end user in an automatic way. Wide usage of continuous integration systems with automatic tests run allows to automate testing process to make sure the functionality is not broken by separate change in a program code. That is why it is important to build suitable testing framework to satisfy needs in continuous testing of specific software.

A source code static analysis tools are become an industrial standard for software quality assurance at early stages in secure software development lifecycle. They are commonly used for detection of program issues and logical errors. Being quality assurance tools by nature they need to satisfy specific requirements such as an analysis precision, completeness and performance. A possibility to introduce bug warnings of a safe code, also known as false positive warnings, set a target for a testing framework to control as true positive warning, as false positive warnings. An acceptance testing of such tools controls behavior of a tool on specific code snippets, which represent as buggy code, as code which has no bugs and issues. At the same time such tools are very complex in implementation details, because consist of general analysis framework, frequently called engine, which propose general analysis techniques such as reaching definitions, live variables, taint analysis and others, and a number of specific wrong program behavior checkers build on top of an engine. Any small change to the engine can broke checkers behavior. That's why it is important to have testing framework which can check and state sanity of the tool during development lifecycle.

In our previous paper we have described a generalized approach for testing static source code analysis tools, which includes Acceptance Testing Framework and Regression Testing System called Report History Server [1].

In this paper we introduce requirements, implementation details, evaluation and limitations of Acceptance Testing Framework for static source code analysis tools based on our experience of development and daily usage of such a framework in industrial development of static source code analysis tools. This paper is organized as follows. Section II describes in detail requirements to such kind of framework, Section III provides overview of existing approaches, Section IV provides an overview of proposed approach. Section V describes in detail implementation of proposed approach, Section VI contains evaluation results of proposed approach, Section VII concludes proposed approach and future directions of development.

II. Requirements to Acceptance Testing Framework

Source code static analysis tools have to check conditions of source code of programs from the point of view of very different rules, which can be applied as industrial or companywide coding standard. Despite of focus for modern static source code analyzers on code security, lack of logical errors and performance, some kind of coding rules applied in companies or industry can contain such requirements to the code as style of indentation, naming conventions, etc. For example, if we take a look to Python programs then source code can contain commentaries of the specific look, such as Shebang [2], encoding of the file [3], company code ownership statement and version or license notes. That's why trying to satisfy needs of testing industrial static source code analyzers such a framework cannot rely on specific comments and code formatting, such as used in most known test cases database Juliet of National Institute for Standardization and Technology of USA [4].

Instead of that we have to have a database of error code snippet describers. Such kind of describers provide all necessary information on test case in a file or set of files with directories structure, separated and independent of language for a source code of target analyzer and target language of analyzed programs. We use specific JSON [5] formatted descriptions of test cases which describe every test case as for erroneous examples, as for clean code examples.

On the other hand we have set a goal to compare tested static source code analyzer with competing ones. That's why we put as a requirement ability to run competing static source code analyzers in one bundle to compare precision, completeness and performance of such tools. That is second requirement.

Next, we need to have solution for different environments such as operating systems and hardware platforms. That's why we set it as one of requirements to the framework.

And, last, but not least, we want to make out Acceptance Testing Framework independent of target language of analyzed programs. It should be suitable for testing analyzers for programming languages C, C++, Java, C#, Python and other languages.

To summarize:

- Independence of target environment, such as hardware and operating system.
- Independence of analyzed programming languages.
- Possibility to check source code snippets without modification of original code even in comments part.
- Possibility to check as erroneous, as clean code examples (true positive and false positive warnings checks).
- Support pretty unlimited number of checkers for coding rules, including, but not limited to formatting and comment styles.
- Possibility to compare different static source code analysis tools.
- Possibility to represent results of analysis in different formats: machine readable (JSON, XML and others), output formatted to represent result on the screen, HTML format, etc. with possibility to extend list of reporting formats on demand.

III. Existing approaches

There are a lot of research papers dedicated to evaluation of static code analysis tools [6, 7, 8]. These works observe behavior of static code analysis tools on selected subset of NIST SAMATE test cases for selected OWASP [9] Top 10 vulnerabilities. But these papers a dedicated to manual evaluation of static code analysis tools and does not solve the problem of automated frameworks implementation. The work [10] attempts to solve the problem of creating automated test suite to evaluate static analysis tools by designing test cases as small code snippets, which automatically in-lined into template program to specific placeholder. The work [11] describes an approach of detecting minimal original test cases from real-world found errors and tries to add code to the original test code snippet to check sensitivity of analysis to paths and call context. The difference of our approach is in common automation of acceptance testing and evaluation system for static source code analysis tools. In this paper we describe technical details and evaluation of proposed approach.

IV. Overview

Acceptance Testing Framework solves problem of evaluating the quality of automatic program analysis tools. The quality is measured by parameters such as: performance, scalability, precision, completeness.

Performance — how fast an analysis tool can provide an analysis result and how much resources it consumes.

Scalability — how analysis time reduces if we providing additional computational resources.

Precision — how precise an analysis result is (small number of false positive warnings or noise).

Completeness — how many true positive warnings issued by a tool in comparison to errors exist in the test suite (number of false negatives — errors has been missed).

To compute such parameters Acceptance Testing Framework allows to run program analysis tool against a limited, manually crafted set of test cases combined in one test suite. Test suite represents behavior of defective and similar to defective programs. The defective one gives rate of true positive warnings should be found and similar to defective gives rate of false positive warnings, which absence is expected. So far the resulting precision and completeness are calculated and evaluated.

As far as precision and completeness are evaluated by Acceptance Testing Framework for program analysis tool, decision about quality could be made. In theory perfect tool has 100 % completeness of test suite (all defects detected) and 100 % precision (no noise and no defect detected on similar to defective code snippets), but such values cannot be achieved at current stage of engineering and have the theoretical limitation of Rice's theorem [12].

There are no strict generally accepted values for performance and scalability as far as these parameters depend on depth, complexity and target of analysis and vary greatly among analysis tools. Moreover, the exact conclusion about the quality of analysis tools directly depends on the test suite. Acceptance Testing Framework doesn't contain built-in features to get performance and scalability on its own for now. Despite this Acceptance Testing Framework could be used in the computation process of these parameters by running program analysis tool against set of different complexity (from low to high) test suites and observe how performance dynamic depends on complexity of test suite or scalability dynamic in the case of additional computational resources involved in computation process.

Test suite could follow company or industrial standards, contain code snippets with security vulnerabilities, code style or leading to a crash errors. In our case test suite follows company standard and together with Acceptance Testing Framework has deployed in continuous integration processes of static analysis tool development in Huawei Russian Research Institute.

V. Design & Implementation

In this section we describe the design and implementation of our framework. We describe it from requirements perspective.

A. Independence of target environment.

To satisfy requirement of an independence of target environment such as hardware and operating system we managed to implement our framework in Python programming language as far as it has Python source code

interpreters for most of industrial operating systems and for most popular hardware platforms.

B. Independence of analyzed programming language.

The framework does not rely somehow on code snippets content by using JSON formatted test case annotations.

C. Possibility to check code snippets without modification of original code, even in comments. Possibility to check as erroneous, as clean code snippets without modification.

We use test case annotation files in JSON format. Test case for Acceptance Testing Framework is a tuple of annotation file and source code snippet. JSON annotation file contains following information:

- Kind of a snippet: does it contains a defect (True Positive) or it is not expected in this code snippet (True Negative).
- Kind of a defect expected to be reported or not reported.
- Description of a test case.
- Skip flag for marking test cases which are not supported, but planned to be supported in future.
- Defect location: filename, line and offset in the line for expected defect.
- Additional service information. For example, if test case designed for specific version of language, to configure analyzer appropriately, or additional field describing the goal of test case to QA engineer or developer.

Such decision allows to keep all this information independent of test cases and needed by Acceptance Testing Framework to configure analysis tools appropriately.

And do not rely somehow on number of test cases, because it is enough to just point the location of file system directory with test suite formatted to be used with Acceptance Testing Framework while running framework and all work related to running analysis tools on the test suite handled by framework itself via traversing directories structure.

D. Possibility to compare different analysis tools.

Acceptance Testing Framework satisfy this requirement by introducing abstract interface *Tool* to run external analysis tool as executable program and get results of analysis in Acceptance Testing Framework internal representation. Having such kind of interface to support of new analysis tool ones need to implement interface *Tool* to convert test case settings from test case annotations to expected arguments of analysis tool and run this tool as external process. We have developed a number of interface implementations for tools, such as PyLint [13], JetBrains PyCharm [14] and eight more tools, which have different paradigm of analysis. For example, PyLint accepts analysis of single file and can be run on every test case separately. PyCharm expects a file system directory and treats it as one project to analyze.

On the other hand analysis results representation of different tools can vary significantly. An implementation of *Tool* interface also responsible for interpretation of external analysis tool results and converting it to Acceptance Testing Framework internal representation. This representation is a kind of map for every test case to analysis result in term of *Passed* or *Failed* state.

Thus all logic of working with analysis tool is encapsulated inside of *Tool* interface implementation.

E. Possibility to represent results of analysis in different formats.

Acceptance Testing Framework provides universal interface *Reporter* which provides one public method *report* accepting internal representation of analysis tool run results. A responsibility of implementation of interface is to issue report in specific format. We have implemented three reporters supported out of the box:

- Output reporter. Represents test suite run results in human readable text format.
- JUnit reporter. Represents test suite run results in JUnit format.
- HTML reporter. Represents test suite run results in format of static web-site with possibility to represent result in different view up to source code snippet of test case.

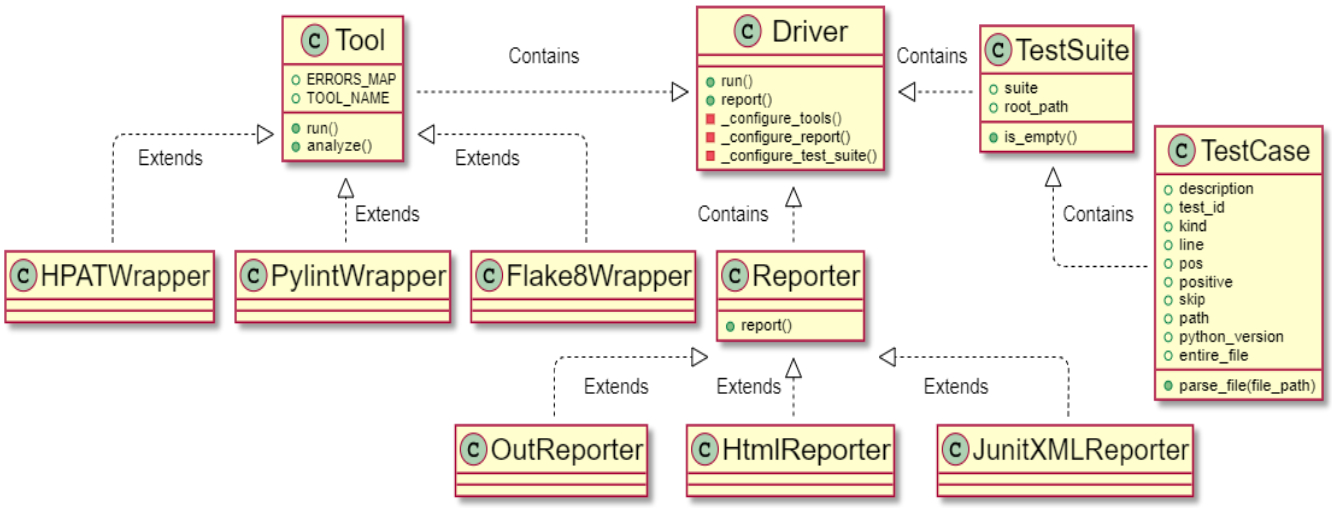

Fig. 1. Acceptance Testing Framework architecture diagram

Architecture diagram of Acceptance Testing Framework is shown on Fig. 1. It consists of following blocks (classes):

- *Driver*. It is entry point of framework. It allows to configure test suite, reporter and tools accordingly to parameters passed to framework on the run.
- *TestSuite* is a collection of *TestCases* which constructed using provided path to test suite directory, where every test case has it's annotation in JSON format and test case source code files directory structure.
- *Tool*. It is an interface representing a tool runner. Instantiations of this interface depends on settings of the framework passed as command line arguments.
- *Reporter*. It is an interface allowing to represent analysis results using unified internal test suite run results representation.

In general, Acceptance Testing Framework is a *Driver*, which responsible for:

- Instantiation of supported analysis tool wrappers, which are implementations of *Tool* interface, accordingly to parameters passed to the *Driver* by user.
- Instantiation of the *Reporter* which will be used to output result of analysis by every tool.
- Running the analysis process to collect analysis result in internal representation form and pass received result to *Reporter*.

VI. Results & Evaluations

This section aims to obtain a classification of tools according to the metrics applied to the results obtained from the execution of the tools against our test suite.

Tested static analysis tools:

- Huawei Python Analysis Tool (HPAT) is a PyCharm plugin with the set of inspections requested by Huawei Python Code Style Guide and Huawei Secure Coding Style Guide.
- Flake8 [15] is an open source tool that glues together pep8 [16], pyflakes [17], mccabe [18], and third-party plugins to check the style and quality of some python code.
- PyLint is an open source tool that checks for errors in Python code, tries to enforce a coding standard and looks for code smells.

The summary of metrics used is:

- True positives rate – TP (correct detections).
- False positive – FP (reporting false error warning).
- Number of vulnerability categories for which the tool was tested.
- Precision (1). Proportion of the total TP detections:

$$TP / (TP + FP) \quad (1)$$

- Recall (2). Ratio of detected vulnerabilities to the number that really exists in the code. Recall is also referred to as the True Positive Rate:

$$TP / (TP + FN) \quad (2)$$

Tab. 1 and Fig. 2 shows a number of vulnerability categories (NVC) for which the tool is tested. HPAT has the biggest value because test suite is developed exactly for satisfying needs of Huawei coding standards.

TABLE I. NUMBER OF VULNERABILITY CATEGORIES

Tool Metric	*HPAT*	*Pylint*	*Flake8*
NVC	68	32	15

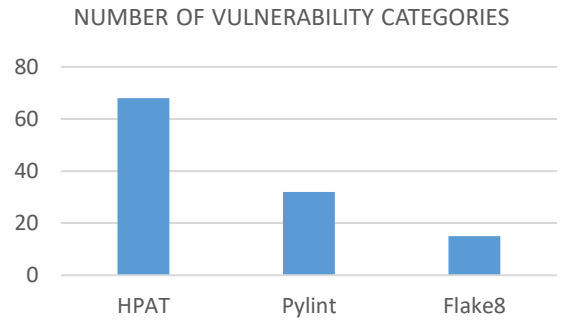

Fig. 2. Number of checked defect types

Tab. 2 and Fig. 3 shows a result of running tools on test suite in terms of true/false positive, true/false negative.

TABLE II. VULNERABILITIES DETECTION. NUMBERS OF TRUE/FALSE POSITIVE, TRUE/FALSE NEGATIVE TEST CASE DETECTION

Tool Metric	*HPAT*	*Pylint*	*Flake8*
TP	695	91	102
FN	0	324	368
FP	0	0	0
TN	591	121	184
Total	1286	536	654

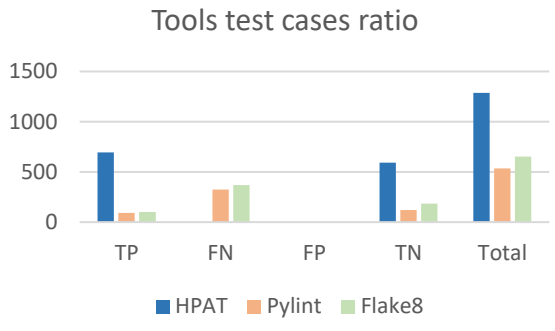

Fig. 3. Test cases ratio obtained by the tools comparison

Tab. 3 and Fig. 4 show metrics results of all tools included in this analysis.

TABLE III. ASSESSMENT RESULTS COMPUTING AND RANKING THE SELECTED METRICS BY TP RATIO

Metric Tool	*TP ratio*	*FP ratio*	*Precision*	*Recall*
HPAT	1	0	1	1
Pylint	0.219	0	1	0.219
Flake8	0.217	0	1	0.217

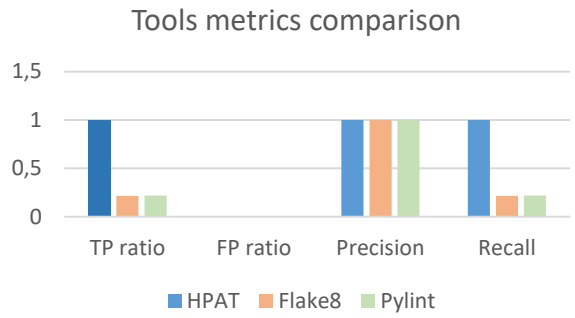

Fig. 4. Metrics obtained by the tools comparison

Implemented framework allows to assess tools on the same testing code base and present relative results.

VII. CONCLUSION

In this paper we were focused on checking quality of static source code analysis tools with help of an automated framework for running such tools against a number of test cases combined in one suite. This approach allows us to control quality of the tool in terms of created erroneous and error free test cases as code snippets on target for analysis programming language. The framework allows to use any kind of test suites if configured well within a profile or manifest in expected format. This approach to testing static source code analysis tools has applied in development process of static source code analysis tools for Python and C/C++ in Huawei Russian Research institute. In future we plan to extend functionality of Acceptance Testing Framework to check non-functional requirements for tools such as time of running, memory consumption and CPU utilization.

REFERENCES

[1] A. Gerasimov, P. Privalov, S. Vladimirov, V. Butkevich, N. Chernova, A. Gorelova. An approach to assuring quality of automatic program analysis tools. Proceedings of 2020 Ivannikov ISP RAS Open Conference (ISPRAS).(Still no reference to printed version)

[2] M. Cooper. Advanced Bash Scripting Guide – Volume 1: An in-depth exploration of the art of shell scripting. (Revision 10) 2019, 589 p.

[3] M.-A. Lemburg, M. von Löwis. PEP-263 – Defining Python Source Code Encodings. 2001.

[4] NIST SAMATE Juliet Test Suite https://samate.nist.gov/SARD/testsuite.php

[5] RFC-8259. The JavaScript Object Notation (JSON) Data Interchange Format, 2017.

[6] H.H. AlBreiki, Q.H. Mahmoud. Evaluation of static analysis tools for software security. IEEE 2014 10th International Conference on Innovations in Information Technology. Al Ain, United Arab Emirates, 2014.

[7] R. Mamood, Q.H. Mahmoud. Evaluation of static nalysis tools for finding vulnerabilitites in Java and C/C++ source code. arXiv:1805.09040, 2018.

[8] T. Hofer. Evaluating static source code analysis tools. Masters thesis. École Polytechnique Fédérale de Lausanne, 2010, pp. 1-74.

[9] OWASP – Open web application security project. https://owasp.org

[10] M. Johns, M. Jodeit. Scanstud: a methodology for systematic, fine-grained, evaluation od static analysis tools. 4th International conference on software testing, verification and validation workshops. Berlin, Germany, 2011, pp. 523-530.

[11] G. Hao, F. Li, W. Huo, Q. Sun, W. Wang, X. Li, W. Zou. Constructing benchmarks for supporting explainable evaluations of static application security testing tools. 2019 International symposium on Theoretical Aspects of Software Engineering, Guilin, China, 2019, pp. 66-72.

[12] H. G. Rice Classes of Recursively Enumerable Sets and Their Decision Problems. Transactions of the American Mathematical Society, 1953, Vol. 74, No. 2, pp. 358-366.

[13] Pylint. https://pypi.org/project/pylint/

[14] JetBrains PyCharm. https://www.jetbrains.com/pycharm/

[15] Flake8. https://pypi.org/project/flake8/

[16] Pep8 – Python style guide checker. https://pypi.org/project/pep8/

[17] Pyflakes. https://github.com/PyCQA/pyflakes

[18] McCabe coomplexity checker https://github.com/PyCQA/mccabe