AFRL-RI-RS-TR-2019-013

SOFTWARE ANALYTICS FOR BIG CODE

UNIVERSITY OF CALIFORNIA

JANUARY 2019

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2019-013 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WILMAR SIFRE
Work Unit Manager

/ S /
STEVEN JOHNS
Chief, Trusted Systems Branch
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
JANUARY 2019	FINAL TECHNICAL REPORT	AUG 2016 – AUG 2018

4. TITLE AND SUBTITLE

SOFTWARE ANALYTICS FOR BIG CODE

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8750-16-2-0286

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Cristina Lopes

5d. PROJECT NUMBER
MUSE

5e. TASK NUMBER
SU

5f. WORK UNIT NUMBER
CI

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The Regents of the University of California
141 Innovation, Suite 250
Irvine, CA, 92697-7600

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2019-013

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report summarizes the technical activities performed by Leidos as the Technical Area 1 and 5 (Evaluation and Infrastructure) lead for DARPA's Mining and Understanding Software Enclaves (MUSE) program. The report describes the technical plans, progress/results, and findings and lessons learned from: corpus collection and curation; benchmark and challenge problem design; metrics and evaluation processes and results; and infrastructure design, implementation, and maintenance.

15. SUBJECT TERMS

Corpora Benchmarks, Open Binary Corpora, Open Source Corpora

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**WILMAR SIFRE**
U	U	U	UU	35	**19b. TELEPHONE NUMBER** (*Include area code*) **N/A**

Contents

ACKNOWLEDGEMENTS iv

SUMMARY 1

1 INTRODUCTION 2
1.1 PROBLEM STATEMENT 2
1.2 CONCRETE GOALS 2

2 G1: CODE DUPLICATION IN GITHUB 4
2.1 METHODS, ASSUMPTIONS, AND PROCEDURES 4
2.2 RESULTS 6

3 G2: STACK OVERFLOW SNIPPETS 9
3.1 FROM QUERY TO USABLE CODE 9
3.1.1 METHODS, ASSUMPTIONS, AND PROCEDURES 9
3.1.2 RESULTS 10
3.2 STACK OVERFLOW IN GITHUB 12
3.2.1 METHODS, ASSUMPTIONS, AND PROCEDURES 12
3.2.2 RESULTS 13

4 G3: THE JAVA BUILD FRAMEWORK 16
4.1 METHODS, ASSUMPTIONS, AND PROCEDURES 16
4.2 RESULTS 17

5 G4: CLONES IN THE TWILIGHT ZONE 20
5.1 METHODS, ASSUMPTIONS, AND PROCEDURES 20
5.2 RESULTS 21

6 PRODUCTS 23
6.1 SOFTWARE 23
6.2 DATASETS 23
6.3 PUBLICATIONS 23

7 CONCLUSIONS 25

8 REFERENCES 26

ACRONYMS 28

List of Figures

1 Analysis pipeline. 4
2 Files per project. 5
3 File-level duplication for entire dataset and excluding small files. 6
4 Map of code duplication. The y-axis is the number of commits per project, the x-axis is the number of files in a project. The value of each tile is the percentage of duplicated files for all projects in the tile. Darker means more clones. 8
5 Sequence of operations for determining usability of snippets. 10
6 Parsable and compilable/runnable rates histogram 11
7 Pipeline for analysis between GitHub and Stack Overflow. 13
8 SourcererJBF pipeline. 16
9 Number of projects built by JBF. 17
10 All build systems vs. JBF, 189,220 projects total. 18
11 Overview of Oreo. 21

List of Tables

1 GitHub Corpus. 5
2 File-Level Duplication. 6
3 Operations performed for each language. 10
4 Summary of results 11
5 Summary of results for C# and Java after single-word snippets removal . . . 11
6 Usability Rates of Top Results from Google (Python only) 12
7 Block-hash similarity 13
8 Token-hash similarity 14
9 SCC Similarity 14
10 Corpus of Java projects. 17
11 Recall and Precision Measurements on BigCloneBench 21

ACKNOWLEDGEMENTS

The PI would like to acknowledge all collaborators in her team, namely: Pedro Martins (post-doc, University of California, Irvine), Rohan Achar (Graduate Research Assistant, University of California, Irvine), Vaibhav Saini (Graduate Research Assistant, University of California, Irvine), Di Yang (Graduate Research Assistant, University of California, Irvine), Prof. Jan Vitek (Northeastern University), Petr Maj (ReactorLabs, Prague), Jakub Zitny (Czech Technical University, Prague), and Hitesh Sajnani (Microsoft, Redmont). Thanks also to the staff at the Institute for Software Research at UC Irvine, namely Janet Ko, Debra Brodbeck, and Anna Chang.

SUMMARY

Goals: The primary goal of this work was to perform analytics on very large software repositories, and report them to the teams and to DARPA. These repositories include a subset of GitHub, and the data from Stack Overflow. Examples of findings we uncovered include the amount and nature of code duplication, how easy/hard it is to build arbitrary projects, and the extent to which mixed natural language / source code datasets can support the discovery, and even synthesis, of programs. This work resulted in the development of tools and techniques that support such analytics at scale.

Main Findings: The three main findings of our work were the following. **(1)** We found that GitHub contains a surprising high level of code duplication, coming from a relatively low number of unique, original files. Specifically, the percentage of *original* files is 60% (Java), 27% (C++), 29% (Python), and 6% (JavaScript); the remaining files are exact duplicates of these. When looking at less than exact-copy, these percentages are even lower. **(2)** We found that Stack Overflow contains a reasonably decent number of code snippets that are usable almost as-is, especially for dynamic languages. Specifically, for Python we found 402,000+ (76%) parsable snippets, of which 135,000+ are runnable as-is; for JavaScript we found 537,000+ (66%) parsable snippets, of which 163,000+ are runnable as-is. We also were able to find many Stack Overflow snippets in use in open source projects hosted in GitHub. **(3)** As for large scale compilation, we developed a build framework that is capable of successfully compiling 190,000+ (54%) projects out of 353,000+ Java projects in our test dataset; this is a significant improvement over using the projects' own build scripts, which results in less than 50,000 projects successfully built.

Approved for Public Release; Distribution Unlimited.

1 INTRODUCTION

1.1 PROBLEM STATEMENT

With the current state of the art in software synthesis, the kinds of programs that can be generated to be provably correct with respect to some formal specification are only a very small percentage of all programs that people want to write. Even when research makes leaps of progress on that front, the fundamental problem of verifying that a formal specification is complete with respect to the *intent* behind the program is essentially unsolvable. Many observations are made afterwards by people, which adds knowledge that is missing from the original specifications. Conversely, implementations found in large code bases, even when they use formal specifications such as types, test cases, or contracts, are likely to lack the expression of important operational and performance properties that may be important for their effective use.

The MUSE Program was based on the belief that the next generation of programming languages and environments need to leverage the large amount of source code and knowledge that is freely available on the Web and, to some extent, automate the search practices that developers use on a daily basis. By explicitly supporting this practice, we may then aim at not just providing programs that fit the purpose, but that are also *the best* at doing it.

The setup for the MUSE program was the following: one of the performers (first Leidos, then Two Six Labs) was tasked with developing an infrastructure for collecting large amounts of projects from open source repositories, which were made available to all the other performers. The collection was made using a relatively brute force approach, as the goal was to get as large of a corpus as possible. However, as is usually the case in all problems involving data mining and machine learning, the results are only as good as the data. Large code bases, however, especially those available in open source repositories were relatively unknown territory. As such, software analytics became a critical component of this program.

For example, in analyzing the existing Java part of the Leidos corpus, we have discovered a relatively large amount of code duplication. Data mining and machine learning efforts need to be informed about this duplication and avoid it, or target it, depending on what they are trying to achieve.

The availability of large amounts of source code and software-related Q&A data poses both opportunities and challenges. Knowing what these repositories contain, and devising the appropriate ontologies to understand them, is a critical first step towards using this data for innovative synthesis tools. This work informs both the performers as well as the DARPA community about what to expect when building software tools using these very large software repositories.

1.2 CONCRETE GOALS

Our work focused on four concrete goals:

G1. Quantification of code duplication in very large repositories of sourse code. We have

conducted a very large scale empirical study on four language ecosystems hosted at GitHub in with the goal of quantifying and qualifying code clones in this large software repository, on which MUSE is based. Our preliminary studies on smaller datasets suggested that there might be a very large amount of code duplication in GitHub, and it was imperative to investigate this further.

G2. Exploration of Stack Overflow code snippets. We conducted two empirical studies to analyze the usability of these snippets, and how they are used in real open source code.

G3. Automatic builds of very large numbers of Java projects. The MUSE program uses open source projects as input data. But many times, we need more than just the source code – we need the compiled versions too. Building hundreds of thousands of projects which we don't control poses major technical challenges pertaining to dependencies. We have developed heuristics and methods for automatically resolving dependencies of Java projects, so that they can be compiled.

G4. Improvement of clone detection tools. Our existing clone detection tool, SourcererCC, does a good job at detecting code clones as long as there is a fair amount of token overlap among the code pairs. This is also true for other code clone detectors. But all these tools start failing fast for code clones that have less lexical and syntactic similarity. We set out to improve the performance of clone detection for these hard-to-detect clones.

The rest of this report explains how each of these goals was accomplished.

Approved for Public Release; Distribution Unlimited.

2 G1: CODE DUPLICATION IN GITHUB

2.1 METHODS, ASSUMPTIONS, AND PROCEDURES

During the execution of this project, we have used our clone detector SourcererCC [9] to conduct a very large empirical study of clones in GitHub within the Java, C++, Python, and JavaScript ecosystems. Here, we summarize the methods and data used for this study. More details can be found in [4].

Our analysis pipeline is outlined in Figure 1. The pipeline starts with local copies of the projects that constitute our corpus. From here, code files are scanned for fact extraction and tokenization. Two of the facts are the hashes of the files and the hashes of the tokens of the files. File hashes identify exact duplicates; token hashes allow catch clones up with minor differences. While permutations of same tokens may have the same hash, they are unlikely. Clones are dominated by exact copies, and we did not observe any such collision in randomly sampled pairs. Files with distinct token hashes are used as input to the near-miss clone detection tool, SourcererCC. While our JavaScript pipeline was developed independently, data formats, database schema and analysis scripts are identical.

The GitHub projects were downloaded using the GHTorrent database and network [2] which contains meta-data such as number of stars, commits, committers, whether projects are forks, main programming language, date of creation, etc., as well as download links. While convenient, GHTorrent has errors: 1.6% of the projects were replicated entries with the same Universal Resource Locator (URL); only the youngest of these was kept for the analysis.

Table 1 gives the size of the different language corpora. We skipped forked projects as forks contain a large amount of code from the original projects, retaining those would skew our findings. Downloading the projects was the most time-consuming step. The order of downloads followed the GHTorrent projects table, which seems to be roughly chronological. Some of the URLs failed to produce valid content. This happened in two cases: when the projects had been deleted, or marked private, and when development for the project happens in branches other than master. Thus, the number of downloaded projects was smaller than the number of URLs in GHTorrent. For each language, the files analyzed were files whose extensions represent source code in the target languages. For Java: `.java`; for

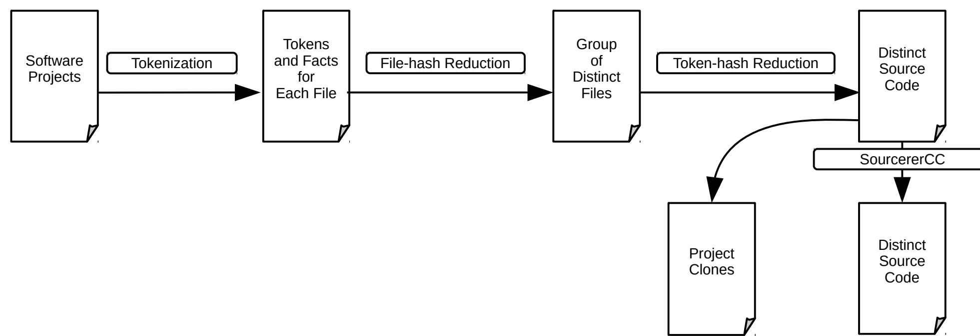


Figure 1: Analysis pipeline.

Table 1: GitHub Corpus.

		Java	C++	Python	JavaScript
Counts	# projects (total)	3,506,219	1,130,879	2,340,845	4,479,173
	# projects (non-fork)	1,859,001	554,008	1,096,246	2,011,875
	# projects (downloaded)	1,481,468	369,440	909,290	1,778,679
	# projects (analyzed)	1,481,468	364,155	893,197	1,755,618
	# files (analyzed)	72,880,615	61,647,575	31,602,780	261,676,091
Medians	Files/project	9 ($\sigma = 600$)	11 ($\sigma = 1304$)	4 ($\sigma = 501$)	6 ($\sigma = 1335$)
	SLOC/file	41 ($\sigma = 552$)	55 ($\sigma = 2019$)	46 ($\sigma = 2196$)	28 ($\sigma = 2736$)
	Stars/project	0 ($\sigma = 71$)	0 ($\sigma = 119$)	0 ($\sigma = 99$)	0 ($\sigma = 324$)
	Commits/project	4 ($\sigma = 336$)	6 ($\sigma = 1493$)	6 ($\sigma = 542$)	6 ($\sigma = 275$)

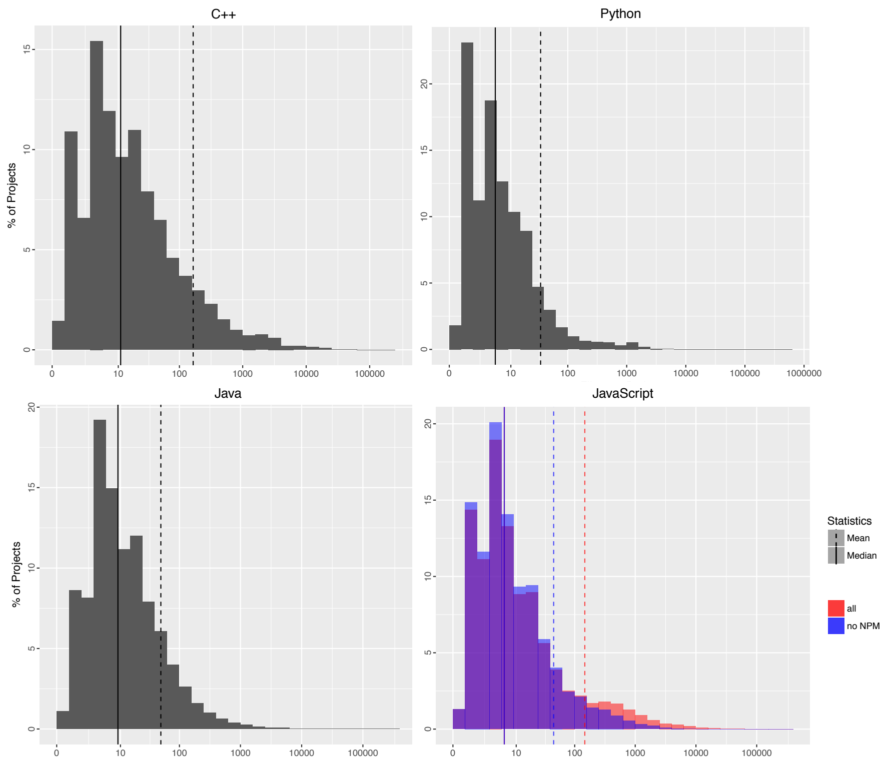


Figure 2: Files per project.

Python: `.py`; for JavaScript: `.js`, for C/C++: `.cpp` `.hpp` `.HPP` `.c` `.h` `.C` `.cc` `.CPP` `.c++` and `.cp`. Some projects did not have any source code with the expected extension, they were excluded.

The medians in Table 1 give additional properties of the corpus, namely the number of files per (non-empty) project, the number of Source Lines of Code (SLOC) per file, the number of stars and the number of commits of the projects. In terms of files per project, Python and JavaScript projects tend to be smaller than Java and C++ projects. C++ files are considerably larger than any others, and JavaScript files are considerably smaller. None of these numbers is surprising. They all confirm the general impression that a large

Table 2: File-Level Duplication.

	Java	C++	Python	JavaScript
Total files	72,880,615	61,647,575	31,602,780	261,676,091
File hashes	43,713,084 (60%)	16,384,801 (27%)	9,157,622 (29%)	15,611,029 (6%)
Token hashes	40,786,858 (56%)	14,425,319 (23%)	8,620,326 (27%)	13,587,850 (5%)
SCC dup files	18,701,593 (26%)	6,200,301 (10%)	2,732,747 (9%)	5,245,470 (2%)
SCC unique files	22,085,265 (30%)	8,225,018 (13%)	5,887,579 (19%)	8,342,380 (3%)

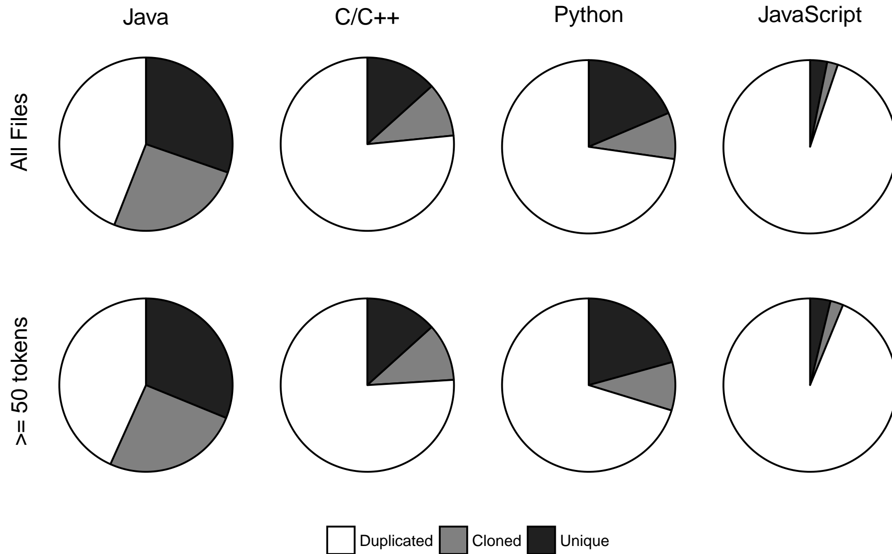


Figure 3: File-level duplication for entire dataset and excluding small files.

number of projects hosted in GitHub are small, not very active, and not very popular. Figure 2 illustrates the basic size-related properties of the projects we analyzed, namely the distribution of files per project. For JavaScript we give data with and without NPM (it is a cause of a large number of clones). Without NPM means that we ignored files downloaded by the Node Package Manager.

2.2 RESULTS

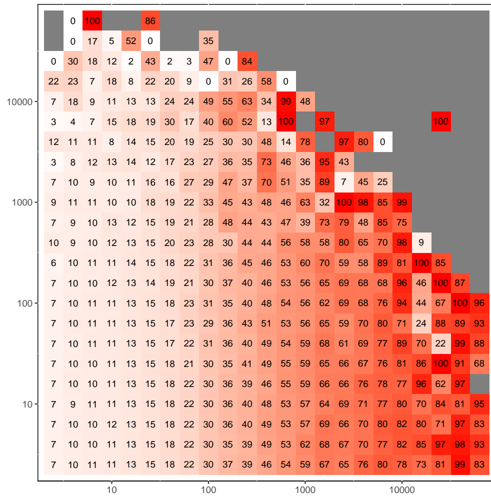
We present the most important results of our study. Comprehensive results can be found in [4]. Table 2 shows a summary of the findings for file-level analysis. "SCC dup files" is the number of files, out of the distinct token-hash files, that SourcererCC has identified as clones; similarly, "SCC unique files" is the number of files for which no clones were detected. Figure 3 (top row) charts the numbers in Table 2. The duplicated files (dark grey) are the files that are duplicate of at least one of the distinct token-hash files (light grey); further, the distinct token-hash files are split between those for which SourcererCC found at least one similar file (cloned files, grey) and those for which SourcererCC did not find any similar file (unique files, in white).

These numbers show a considerable amount of code duplication, both exact copies of the files (file hashes), exact copies of the files' tokens (token hashes), and near-duplicates of files (SourcererCC). The amount of duplication varies with the language: the JavaScript ecosystem contains the largest amount of duplication, with 94% of files being file-hash clones of the other 6%; the Java ecosystem contains the smallest amount, but even for Java, 40% of the files are duplicates; the C++ and Python ecosystems have 73% and 71% copies, respectively. As for near-duplicates, Java contains the largest percentage: 46% of the files are near-duplicate clones. The ratio of near-miss clones is 43% for Java, 39% for JavaScript, and 32% for Python.

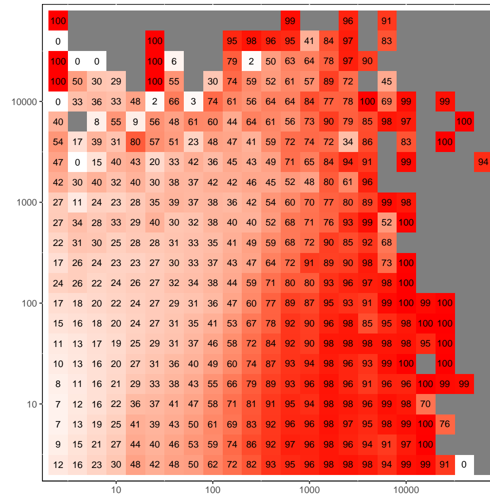
The heatmaps in Figure 4 were produced using the number of commits shown in Table 1, the number of files in each project, and the file hashes. The heat intensity corresponds to the ratio of file hashes clones over total files for each cell.

Duplication can come in many flavors. Specifically, it could be evenly or unevenly distributed among all token hashes. We found these distributions to be highly skewed towards small groups of files. In Java 1.5M groups of files with the same token-hash have either 2 or 3 files in them; the number of token hash-equal groups with more than 100 files is minuscule. The same observation holds for the other languages. Another interesting piece of information about clone groups is given by the largest extreme. In Python, the largest group of file-hash clones has over 2.5M files. In Java, the largest group of SourcererCCClones has over 65K files. In the next section we show which files these are.

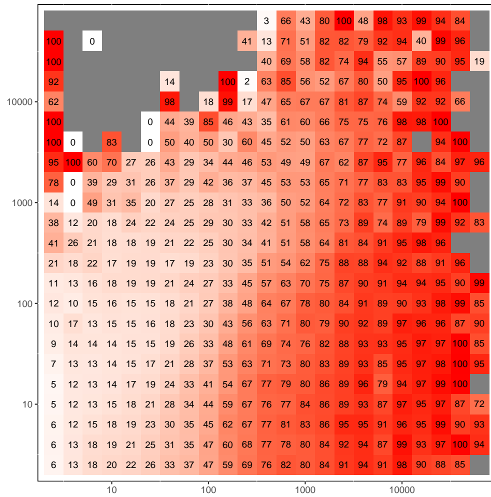
Approved for Public Release; Distribution Unlimited.



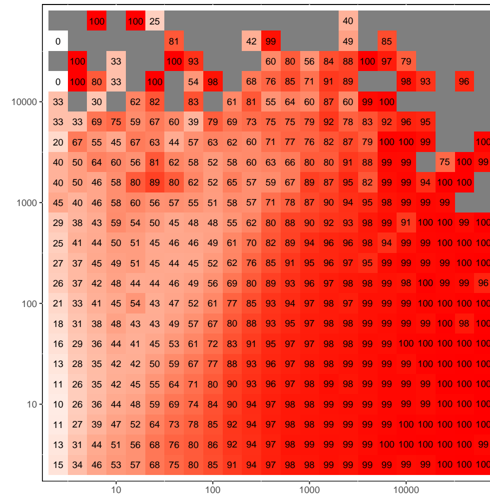
Java



Python



C++



JavaScript

Figure 4: Map of code duplication. The y-axis is the number of commits per project, the x-axis is the number of files in a project. The value of each tile is the percentage of duplicated files for all projects in the tile. Darker means more clones.

3 G2: STACK OVERFLOW SNIPPETS

We conducted two studies of Stack Overflow (SO) code snippets. The first study investigated how many snippets in SO are usable as-is, and whether search engines find the usable ones. The second study investigated how programmers use these snippets in open source projects. We briefly describe the goals and methods for these two studies.

3.1 FROM QUERY TO USABLE CODE

3.1.1 METHODS, ASSUMPTIONS, AND PROCEDURES

Over the years, SO has accumulated an impressive amount of programming knowledge consisting of snippets of code together with relevant natural language explanations. Besides being useful for developers, SO can potentially be used as a knowledge base for tools that automatically combine snippets of code in order to obtain more complex behavior. Moreover those more complex snippets could be retrieved by matches on the natural language (i.e. non-coding information) that enriches the small snippets in SO.

In pursuing this goal, the first challenge one faces is whether, and to what extent, the existing snippets of code that are suggested by Web search results are usable as-is. If there are not enough usable snippets of code, the process of repairing them automatically for further composition may be out of reach. This document presents research in this direction, by showing the results of our investigation of the following questions: (1) How usable are the SO code snippets? (2) When using Web search engines for matching on the natural language questions and answers around the snippets, what percentage of the top results contain usable code snippets?

In order to compare the usability of different pieces of code, we need to define what usability is in the first place. We classified snippets of code based on the effort that would (potentially) be required by a program generation tool to use the snippet as-is. Usability is therefore defined based on the standard steps of parsing, compiling and running source code. For each of these steps, if the source code passes, the more likely it is that the tool can use it with minimum effort. Given this definition of usability, there are situations where a snippet that does not parse is more useful than the one that runs, but passing these steps assures us of important characteristics of the snippet, such as the code being syntactically and structurally correct, or all the dependencies being readily available, which are of surmount importance for automation.

We first study the percentages of parsable, compilable and runnable (where these steps apply) snippets for each of the four most popular programming languages (C#, Java, JavaScript, and Python). Next, we focused on the best performing language (Python) and conducted a 3-step qualitative analysis to see if the runnable snippets can actually answer questions correctly and completely. Finally, in order to close the circle, we use Google Search in order to find out the extent to which the SO snippets suggested by the top Google results are usable. If we are able to use informal Google queries to locate relevant pieces of code in Stack Overflow, then many tasks become automatable from those informal queries.

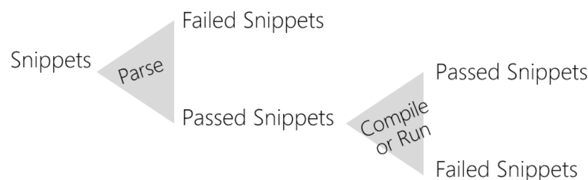


Figure 5: Sequence of operations for determining usability of snippets.

Dataset. The snippets were extracted from the dump available at the Stack Exchange data dump site[1]. In this study, we only included snippets found in all **accepted answers**, assuming that it is very likely that this answer resolved the original problem. For all posts for a language we were interested in, we used the markdown `<code>` to extract the code snippets from the field `Body`.

Data Processing. In Table 3, we present the operations we performed to analyze and rate each language. All the snippets from all languages were parsed, but depending on the static or dynamic nature of the language we either compiled it and analyzed the (possible) errors, or ran the language (below we detail these processes).

Table 3: Operations performed for each language.

Operation	C#	Java	JavaScript	Python
Parse	x	x	x	x
Compile	x	x		
Run			x	x

Figure 5 shows the order in which these operations are performed. We compile (or run) only those snippets which passed parsing, since snippets which are unparsable have syntactic errors and therefore are also non-compilable/non-runnable. More details related to how we processed snippets in each language are given in [13].

Google Queries. We explored the overlap between Google search results and the usable Python snippets. Specifically, we check if the top results from Google for several queries contain parsable or runnable snippets, as well as these snippets' overall quality. The methodology was as follows. We selected 100 programming related questions from SO's highest voted questions about Python, and use them as queries via the Google search API. We add the constraint "site:stackoverflow.com" and the keyword "Python" in the queries. Moreover, because our database was downloaded in April 2014, we also add a date range restriction.

3.1.2 RESULTS

Table 4 and Figure 6 show the summary of usability results of all the snippets. A total of 3M code snippets were analyzed. Python and JavaScript proved to be the languages for which the most code snippets are usable: 537,767 (65.88%) JavaScript snippets are parsable

[1]https://archive.org/details/stackexchange, obtained on April 2014.

Table 4: Summary of results

	C#	Java	JavaScript	Python
Total Snippets	810,829	914,974	816,227	527,774
Parsable	129,727 (16.00%)	35,619 (3.89%)	537,767 (65.88%)	402,249 (76.22%)
Compilable	986 (0.12%)	9,177(1.00%)	–	–
Runnable	–	–	163,247 (20.00%)	135,147 (25.61%)

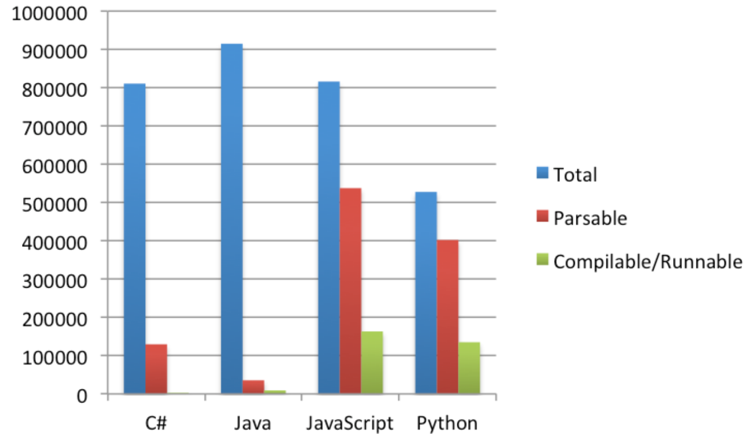


Figure 6: Parsable and compilable/runnable rates histogram

and 163,247 (20.00%) of them are runnable; for Python, 402,249 (76.22%) are parsable and 135,147 (25.61%) are runnable. Conversely, Java and C# proved to be the languages with the lowest usability rate: 129,727 (16.00%) C# snippets are parsable but only 986 (0.12%) of them are compilable; for Java, only 35,619 (3.89%) are parsable and 9,177 (1.00%) compile.

As a result of finding such low parsable and compilable rates for Java and C#, we removed Java and C# snippets that only contained single words (i.e. tokens without non-alphabetical characters). The rationale behind this step was to that a single word in C# or Java is too insignificant a candidate for composability; by ignoring those snippets we might improve the usability rates for these two languages. We then parsed and compiled the remaining snippets, the results of which are shown in Table 5. We see that the rates of usability improve for both languages, and for both parsing and compilation. For Java, the parsable rate increases from 3.89% to 6.22%, and the compilable rate increases from 1.00% to 1.60%. For C#, the parsable rate increases from 16.00% to 25.18%, and the compilable rate increases from 0.12% to 0.19%.

Table 5: Summary of results for C# and Java after single-word snippets removal

	C#	Java
Total snippets after removal	514,992	572,742
Parsable	129,691 (25.18%)	35,619 (6.22%)
Compilable	986 (0.19%)	9,177 (1.60%)

Approved for Public Release; Distribution Unlimited.

Table 6: Usability Rates of Top Results from Google (Python only)

	Parsable	Runnable
Top 1	78.1%	30.8%
Top 10	77.9%	29.3%

As for Google queries, the accepted answers' usability rates of the Top 1 and Top 10 results from Google are shown in Table 6. They are high. As described above, we found that the usability rates of all the Python snippets in SO are 76% parsable and 25% runnable. The top results on 100 queries to Google on the same SO data have usability rates above those averages. Moreover, the Top 1 results have an even higher usability rate than the Top 10 results. Also, we find that 33.7% of Top 1 results and 32.5% of Top 10 results are multiple line snippets. Both higher than the average of 30%. So, both from our usability perspective and qualitative analysis perspective, the Google Top 10 search results are better than average, and the Top 1 results are the best.

Expecting snippets to be runnable as-is may be too strong of a constraint. Parsable snippets seem to be a much more fertile ground as the base for future automatic code generation. Given our analysis of the causes of runtime errors, it seems it should be possible to repair a large percentage of them automatically. For example for Python, missing symbol names often indicate a piece of information that needs to come from elsewhere – another snippet, or some default initialization.

3.2 STACK OVERFLOW IN GITHUB

3.2.1 METHODS, ASSUMPTIONS, AND PROCEDURES

In this study, our goal was to investigate and understand how much the snippets obtained from SO are used in GitHub projects. We opertionalized this problem as pieces of source code that exist in both sides, and we searched for cloning and repetition as a measure of equal information presented in both places. How much of the knowledge base, represented as source code, is shared between SO and GitHub? If SO and GitHub have overlapping source code, is this copy literal or does it suffer adaptations? And are these adaptations, if they exist, specializations required by the idiosyncrasies of the target or by the idiosyncrasies or the programmer, or both?

To answer these questions we performed intra and inter code duplication analysis on GitHub and SO. We uncover and document code duplicates in 909k Python projects from GitHub, which contain 292M function definitions in GitHub and 1.9M snippets in SO. Our choice of language was driven by popularity and our study described in the previous section [13], which shows Python snippets in SO having one of the highest usability rates among the popular languages.

Figure 7 shows the main steps in the analysis process. The pipeline starts by extracting blocks from GitHub projects and SO posts. Blocks from both origins are then scanned to obtain tokens and other relevant information (a process we call tokenization from now on).

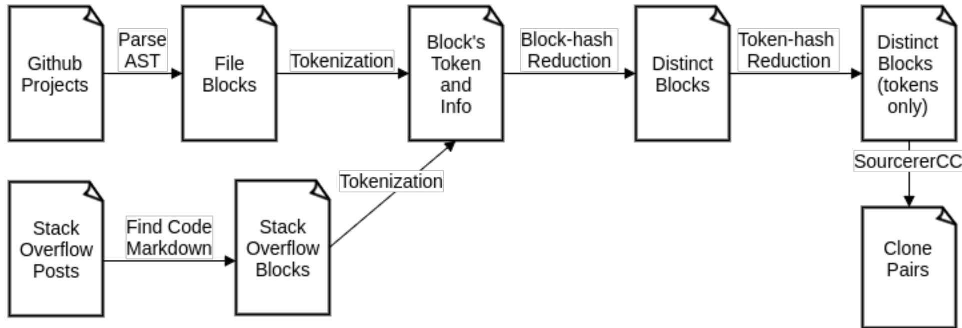


Figure 7: Pipeline for analysis between GitHub and Stack Overflow.

Table 7: Block-hash similarity

	GitHub	SO
Total blocks	290,742,628	1,954,025
Distinct block-hashes	40,098,522	1,929,411
Common distinct block-hashes	1,566	1,566
Common blocks	60,962	2,091

In this analysis process, we provide three levels of similarity: a hash on the block, a hash on tokens, and an 80% token similarity. These capture the case where entire blocks are copied as-is, smaller changes are made in spacing or comments, and more meaningful edits are applied to the code. Moreover, all the similarity analyses are done for intra-GitHub, intra-SO, and inter GitHub and SO. More details can be found in [14].

3.2.2 RESULTS

The main results are presented here; more findings can be found [14]. We provide three types of analysis using, first, hash values on the blocks (block-hash), second, token hash on the blocks' source code (token-hash) and third, partial clones using SourcererCC. This provides different degrees of similarity for blocks: on the first we compare for perfect equality, on the second we filter glueing syntactic elements (spaces, tabs, terminal symbols, etc.), and on the third we allow some divergence.

Block-hash Similarity

The results for block-level hashing can be seen in Table 7. For hash analysis, we start by reducing the total group of blocks to a distinct set of block-level hashes. This set, shown on the second row of the table, represents the number of distinct pieces of code on the datasets. For GitHub, out of the 290M blocks there are only 40M distinct hashes, meaning that block-level code duplication is intense: 86% of blocks have the same exact code as the other 14%. This large amount of code duplication in open source project repositories has been observed before. For SO, the numbers are considerably smaller, with an almost absence of block duplication; only 1.3% of the blocks have the same code as the other 98.7%.

Table 8: Token-hash similarity

	GitHub	SO
Total # blocks	290,742,628	1,954,025
Distinct token-hashes	35,894,897	1,890,565
Common distinct token-hashes	9,044	9,044
Common blocks	3,839,019	13,747

Table 9: SCC Similarity

	GitHub	SO
Distinct token hashes	35,894,897	1,890,565
SCC-dup	13,363,759	297,554
Common	405,393	35,098

Next, we make the intersection of the distinct hashes in both datasets, obtaining the common distinct hashes between GitHub and SO. That number is shown in the third row: 1,566. This is a very small percentage of the distinct hashes in both datasets.

Finally, in row four we count all the blocks whose hashes belong to the common hashes. These are the blocks of code that exist in their exact form, including formatting and whitespace, in both GitHub and SO. The percentages are very small, less than 1% in both cases.

Token-hash Similarity

The results for token-hash analysis are presented in Table 8. Not surprisingly, when formatting and whitespace are ignored, the code duplication in each dataset increases slightly, i.e. the number of distinct token hashes is smaller than the number of distinct block hashes (compare second rows of Tables 7 and 8).

For the same reason, the common distinct token hashes between GitHub and SO is considerably larger than the common distinct block hashes (compare third rows of Tables 7 and 8). But the percentage of distinct hashes that are common to both datasets is still very small.

Interestingly, the number of blocks in GitHub whose token hashes are in the common set is above 1% (see row four). While small, it is remarkable that so many Python functions in GitHub projects, almost 4M, have the exact same tokens as snippets of Python code found in SO.

SourcererCC Similarity

The analysis here is slighly different than the previous two: we narrow the analysis only to the universe of blocks that have distinct token hashes, those counted in the second row of Table 8. The rationale is that two files with the same token-hashes will be detected as clones by Sourcerer, and therefore it suffices to process only one representative of each group of blocks with the same token hash.

Approved for Public Release; Distribution Unlimited.

The results are presented in Table 9. The second row, SCC-dup, shows the number of blocks in each dataset that have at least one similar block in the same dataset – only within the universe of distinct token hashes. The amount of near-duplication is considerably high in GitHub (roughly, 37%), but less in SO (roughly 16%).

The third row shows the number of blocks that are similar between datasets – again, only within the universe of distinct token hashes. More than 405k (1.1%) of the blocks in GitHub are similar to blocks in SO, and 2% of blocks in SO are similar to blocks in GitHub. This means that 35,098 distinct blocks found in SO can be found in very similar form in GitHub. The number is considerably larger than the common distinct token-hashes in Table 8.

Approved for Public Release; Distribution Unlimited.

4 G3: THE JAVA BUILD FRAMEWORK

4.1 METHODS, ASSUMPTIONS, AND PROCEDURES

Scaling up compilation, testing, and execution to hunreds of thousands of projects from Internet repositories is a daunting task. In order to do that, we need automated techniques for resolving dependencies (compilation), for finding and driving test suites (testing), and for producing input and workflows (execution). These three types of automation are all challenging in different ways, with the last one (execution) being the most difficult one. We developed a method and tool for tackling the first challenge: large-scale compilation. Our tool is called the Java Build Framework (JBF). Two elements are at the core of JBF: (1) a very large repository of existing `JAR` files, and (2) a technique of dependency resolution that identifies the external types in the projects and maps them to the most appropriate `JAR`.

An overview of the entire process that constitutes JBF can be seen in Figure 8. JBF is split into two main tasks, namely management of dependencies and project compilation, and each of these is further split into sub-tasks.

Starting with dependency management, the first task on the pipeline consists of collecting `JAR` files from the projects. After that, JBF creates an inverted index of Fully Qualified Names (FQN) of `Java` classes and interfaces available in a `JAR`. This is done before any compilation takes place.

The compilation process in JBF is done in up to two rounds for each project. In a first round, an attempt is made to compile it without specifying any external dependencies, just relying on the `Java` standard libraries. The ones that succeed are marked as so; the ones that fail go through a second round of processing. JBF detects compilation errors related to encoding problems and to missing external dependencies. Both are solved through inserting specific directives in our compilation scripts. For resolving external dependencies, we use the index created in the initial phase. A second round of compilation starts whose results are final: the projects that compile are marked as successes, the failures as failures, and the results of this round together with the results of the first round determine the overall effectiveness of JBF. Details of how JBF works can be found in [5].

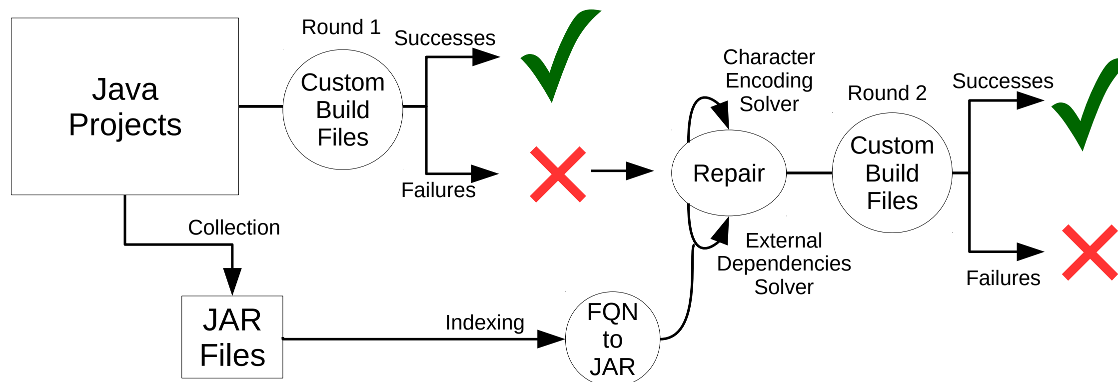


Figure 8: SourcererJBF pipeline.

Table 10: Corpus of Java projects.

# projects (total)	3,506,219
# projects (non-fork)	1,859,001
# URLs processed	631,390
# projects (downloaded)	479,113
# projects (excluding Android)	353,709
# jars	245,648
# FQNs	8,164,106

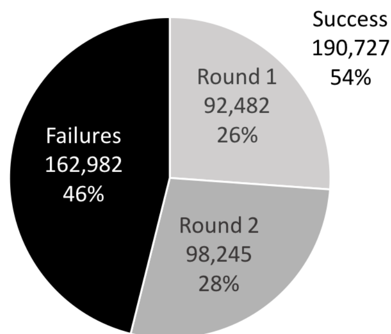


Figure 9: Number of projects built by JBF.

Dataset. We used the Java dataset collected from GitHub (see Section 2). Table 10 shows the information regarding the size of the language corpora in our analysis. We skipped forked projects, because they represent directly cloned projects and usually contain a large amount of code from the original projects; having explicit forks in the dataset for this study could skew the findings.

4.2 RESULTS

Between the two rounds of compilation, JBF compiled more than half the projects, 190,727, which corresponds to 54%. From all the projects that JBF was capable of building, around half built after the first round of compilation, meaning they did not require any kind of special assistance. The other half required the assistance of the error repair mechanism, which proved successful for 98,245 projects. The exact values of the number of projects built after the two rounds can be seen in Figure 9.

To place these results in perspective, we compared the effectiveness of JBF with the effectiveness of using the build scripts included in the projects, for the projects that included build files. Figure 10 shows this comparative results. Both the build frameworks and JBF successfully built 58,785 projects. The build frameworks, on their own, built 47,188 projects, while JBF, on its own, built 28,141 projects. Analyzing all the projects, the successes and failures shared by JBF and the build frameworks cover around two thirds of all the projects; for the other third, the build frameworks perform more effectively.

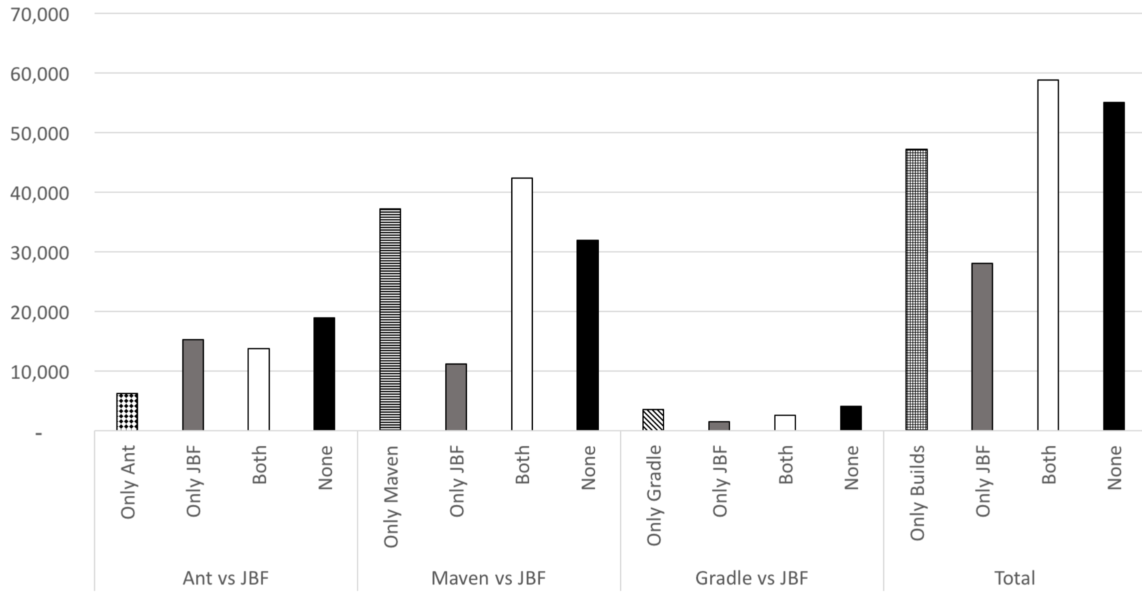


Figure 10: All build systems vs. JBF, 189,220 projects total.

When comparing JBF with existing build systems we must make some important notes:

First, in the results presented in Figure 10, we are neglecting a large number of projects that simply do not contain build files. Within all the 353,709 projects we obtained, if we used build files we would be able to build 105,973 projects, whereas with JBF we were capable of building 190,727 projects, out of which 103,801 projects contain no build information.

Second, the build frameworks analyzed are in their essence scripts that allow a possible infinite number of actions on a large number of domains. They can remove, add or move information in the system or connect to online servers. It is easy to see the security holes that running a large number of unknown script creates, and these should not be taken lightly.

Third, the utility of building software is in the access of the produced runnable elements, in our case class files. But the use of unknown scripts can - and often does - cripple this advantage. We do not know where the targets are being allocated, or if the script is creating a Java ARchive (JAR) file instead of class files.

Fourth, the notion of success is dependent only on the full execution of the build framework, not on its intrinsic actions. This means a script can be successful but everything it does is printing the famous 'Hello World!'. On efforts of building on scale this introduces an uncertainty that easily becomes unreasonable. For example, when we provide the results of the build scripts in the previous sections these represent successes from the perspective of the operating system (the process terminated without an error code), not from the perspective of the inherent compilation task. This problem is of hard leverage. With JBF, "success" means that bytecodes were produced.

Fifth, and related with the previous two, the time it takes to run a build script is as much of a mystery as its contents. The building of the 353,709 `Java` projects using JBF required a median of 7.5 seconds per thread, where each thread was responsible for the complete

process from the management and extraction of the archived files to the compilation and any necessary clean-ups or error recoveries. The actual compile time (a sub-task of each thread) had per project a median of 2.3 seconds. The equivalent times for the building frameworks were 20 seconds in median, and just to compile 7.8 seconds in median. The times, especially the compile time, are substantially longer than JBF times. On universes of hundreds of thousands of projects this is relevant as it represents differences of weeks between one strategy or the other.

Taking these considerations into account, the value of JBF for obtaining bytecode for a very collection of projects becomes much more clear.

Approved for Public Release; Distribution Unlimited.

5 G4: CLONES IN THE TWILIGHT ZONE

One of our most important contributions to the MUSE program was our clone detection tool, SourcererCC. However, we are well aware of its limitations. Therefore, we conducted some research in order to improve the detection of harder-to-detect clones. Our second generation of clone detection is called Oreo, and it is explained here.

5.1 METHODS, ASSUMPTIONS, AND PROCEDURES

There are four broad categories of clone detection approaches, ranging from easy-to-detect to hard-to-detect clones: textual similarity, lexical similarity, syntactic similarity, and semantic similarity. The literature refers to them as the four commonly accepted types of clones [1, 6]:

- Type-1 (textual similarity): Identical code fragments, except for differences in white-space, layout and comments.
- Type-2 (lexical, or token-based, similarity): Identical code fragments, except for differences in identifier names and literal values.
- Type-3 (syntactic similarity): Syntactically similar code fragments that differ at the statement level. The fragments have statements added, modified and/or removed with respect to each other.
- Type-4 (semantic similarity): Code fragments that are semantically similar in terms of what they do, but possibly different in how they do it. This type of clones may have little or no lexical or syntactic similarity between them. An extreme example of exact semantic similarity that has almost no syntactic similarity, is that of two sort functions, one implementing bubble sort and the other implementing selection sort.

Clone detectors use a variety of signals from the code (text, tokens, syntactic features, program dependency graphs, etc.) and tend to aim for detecting specific types of clones, usually up to Type-3. Very few of them attempt at detecting pure Type-4 clones, since it requires analysis of the actual behavior – a hard problem, in general. Starting at Type-3 and onwards, however, lies a spectrum of clones that, although still within the reach of automatic clone detection, are increasingly hard to detect. Reflecting the vastness of this spectrum, the popular clone benchmark BigCloneBench [10] includes subcategories between Type-3 and Type-4, namely Very Strongly Type-3, Strongly Type-3, Moderately Type-3, and Weakly Type-3, which merges with Type-4.

Clones that are moderately Type-3 and onward fall in the *Twilight Zone* of clone detection: reported precision and recall of existing clone detectors drop dramatically for them. For this reason, they are the topic of our work. Our goal is to improve the performance of clone detection for these hard-to-detect clones.

The goals driving the design of Oreo, our new clone detector, are twofold: (1) we want to be able to detect clones in the Twilight Zone without hurting accuracy, and (2) we want to

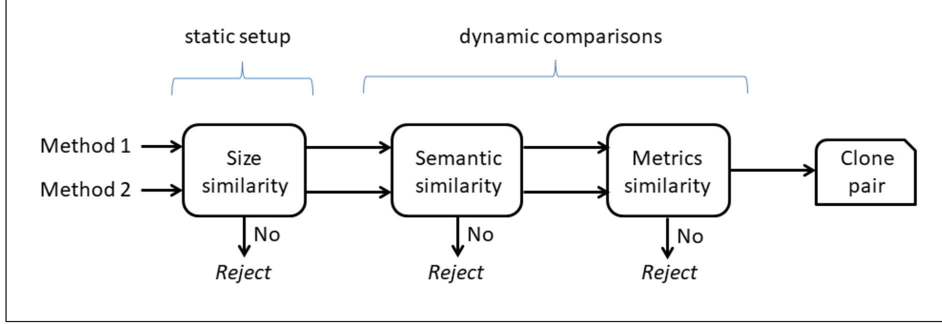


Figure 11: Overview of Oreo.

be able to process very large datasets consisting of hundreds of millions of methods. In order to accomplish the first goal, we introduce the concept of semantic signature based on actions performed by that method, followed by an analysis of the methods' software metrics. In order to accomplish the second goal, we first use a simple size-based heuristic that eliminates a large number of unlikely clone pairs. Additionally, the use of semantic signatures also allows us to eliminate unlikely clone pairs early on, leaving the metrics analysis to only the most likely clone pairs. Figure 11 gives an overview of Oreo. The metrics similarity is accomplished with a machine learning model. More details can be found in [8].

5.2 RESULTS

We compare Oreo's detection performance against the latest versions of the four publicly available clone detection tools, namely: SourcererCC [9], NiCad [7], CloneWorks [11], and Deckard [3]. As Type-1 and Type-2 clones are relatively easy to detect, we focus primarily on Type-3 clone detectors. The configurations of these tools were based on our discussions with their developers, and also the configurations suggested in [12].

Table 11 summarizes the recall and precision for all tools. The recall numbers are summarized per clone category. The numbers in the parenthesis next to the category titles show the number of manually tagged clone pairs for that category in the benchmark dataset. Each clone category has two columns under it, tilted "%", where we show the recall percentage and "#", where we show the number of manually tagged clones detected for that category by each tool. The best recall numbers are presented in *bold typeface*. We note that we couldn't run Deckard on the BigCloneEval as Deckard produced more than 400G of clone pairs and

Table 11: Recall and Precision Measurements on BigCloneBench

Tool	Recall												Precision
	T1 (35,802)		T2 (4,577)		VST3 (4,156)		ST3 (15,031)		MT3 (80,023)		WT3/T4 (7,804,868)		
	%	#	%	#	%	#	%	#	%	#	%	#	Sample=400
Oreo	**100**	**35,798**	**99**	**4,547**	**100**	**4,139**	89	13,391	**30**	**23,834**	0.7	57,273	89.5%
SourcererCC	100	35,797	97	4,462	93	3,871	60	9,099	5	4,187	0	2,005	97.8%
CloneWorks	100	35,777	99	4,544	98	4,090	**93**	**13,976**	3	2,700	0	35	98.7%
NiCad	100	35,769	99	4,541	98	4,091	93	13,910	0.8	671	0	12	**99%**
Deckard	60	21,481	58	2,655	62	2,577	31	4,660	12	9,603	**1**	**780,487**	34.8%

BigCloneEval failed to process this huge amount of data. The recall numbers shown for Deckard are taken from SourcererCC's paper [9], where the authors evaluated Deckard's recall on BigCloneBench. The total number of clone pairs are not available for Deckard, and for this reason, we calculated them based on the reported percentage values.

As Table 11 shows, Oreo performs better than every other tool on most of the clone categories, except for ST3 and WT3/T4 categories. CloneWorks performs the best on ST3 and Deckard performs the best on WT3/T4. Performance of Oreo is significantly better than other tools on the hard-to-detect clone categories like MT3 and WT3/T4, where Oreo detects one to two orders of magnitude more clone pairs than SourcererCC, CloneWorks, and NiCad. This is expected as these tools are not designed to detect harder-to-get clones in the Twilight Zone. The recall numbers are very encouraging as they show that beside detecting easier to find clones such as T1, T2, and VST3, Oreo has the capability of detecting clones that are hardly detected by other tools. In future, we would like to investigate deeper to understand why Oreo did not perform as well as Nicad or CloneWorks on ST3 category.

The precision of Oreo is 89.5%. Deckard's precision is the lowest at 34.8% and Nicad's precision is the highest at 99%. While the precision of Oreo is lower than the other three state of the art tools, it is important to note that Oreo pushes the boundaries of clone detection to the categories where other tools have almost negligible performance.

The recall and precision experiments demonstrate that Oreo is an accurate clone detector capable of detecting clones in Type-1, Type-2, Type-3 and in the Twilight Zone. Also, note that Oreo is trained using the clone pairs produced by SourcererCC. As SourcererCC does not perform well on hard-to-detect categories like ST3, MT3, and WT3/T4, our current training dataset lacked such examples. To address this issue, in future we will train Oreo with an ensemble of state of the art clone detectors.

6 PRODUCTS

As a result of this project several products were developed and/or improved. We list them here.

6.1 SOFTWARE

The following items were delivered to Two Six Labs for integration:

- Pipeline for detecting code clones, including SourcererCC. `http://mondego.ics.uci.edu/projects/dejavu/`
- The Java Build Framework. `http://mondego.ics.uci.edu/projects/jbf/`

The following items were developed and are available as open source:

- Oreo. https://github.com/Mondego/oreo

6.2 DATASETS

- DéjàVu, a map of code clones in GitHub. Available at `http://mondego.ics.uci.edu/projects/dejavu/`
- 50K-C, a dataset consisting of 50,000 Java projects, with source and bytecode. Available at `http://mondego.ics.uci.edu/projects/jbf/`

6.3 PUBLICATIONS

The following publications were produced as a result of this project.

- Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. DéjàVu: A Map of Code Duplicates on GitHub. Proc. ACM Program. Lang., 1(OOPSLA):84, 28 pages, October 2017.
- Pedro Martins, Rohan Achar, and Cristina V. Lopes. The Java Build Framework: Large Scale Compilation. Technical Report UCI-ISR-18-3, Institute for Software Research, University of California, Irvine, 2018.
- Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. Oreo: Detection of Clones in the Twilight Zone. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, pages 354–365, New York, NY, USA, 2018. ACM.

Approved for Public Release; Distribution Unlimited.

- Di Yang, Aftab Hussain, and Cristina Videira Lopes. From query to usable code: An analysis of Stack OverFlow code snippets. In Proceedings of the 13th International Conference on Mining Software Repositories, MSR'16, pages 391–402, New York, NY, USA, 2016. ACM.
- Di Yang, Pedro Martins, Vaibhav Saini, and Cristina Lopes. Stack OverFlow in GitHub: Any snippets there? In Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17, pages 280-290, Piscataway, NJ, USA, 2017. IEEE Press.

7 CONCLUSIONS

The availability of large amounts of source code and software-related Q&A data poses both opportunities and challenges. Knowing what these repositories contain, and devising the appropriate ontologies to understand them, is a critical first step towards using this data for innovative applications such synthesis tools. This work informed both the DARPA MUSE performers as well as the reserarch community about what to expect when building software tools using these very large software repositories.

One of the major findings of our work was the amount of code duplication that exists in open source projects. While we expected duplication, we were surprised by how much duplication we found in GitHub. It is essential that users of these large code bases are aware of this phenomenon, so to avoid skewing results and wasting computing time. It is worth mentioning that our study gained considerable attention in the popular media and blogshpere.

Our studies of Stack Overflow showed that SO is, indeed, a viable and valuable source of rich information for software engineering tools, and that there has a good potential to bridge the semantic gap between natural language queries and code. Moreover, we found another interesting bridge between SO and GitHub by tracking the SO snippets that find their way to real code.

The Java Build Framework we developed helps us compile large numbers of Java projects that may or may not contain build scripts. This is very important to obtain bytecode. Bytecode is essential for executing the code, which may be necessary for advanced software tools that leverage large amounts of open source code.

Finally, we started to push the boundaries of clone deteciton into harder-to-detect clones whose syntactic and semantic similarity is tenuous. While much more needs to be done, our Oreo clone detector shows good potential to expand what can be reliably detected.

8 REFERENCES

[1] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, Sept 2007.

[2] Georgios Gousios. The GHTorrent dataset and tool suite. In *Working Conference on Mining Software Repositories (MSR)*, 2013.

[3] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[4] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.*, 1(OOPSLA):84:1–84:28, October 2017.

[5] Pedro Martins, Rohan Achar, and Cristina V. Lopes. The Java Build Framework: Large Scale Compilation. Technical Report UCI-ISR-18-3, Institute for Software Research, University of California, Irvine, 2018.

[6] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *Technical Report, Queen's University at Kingston*, 2007.

[7] Chanchal K Roy and James R Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC08)*, pages 172–181. IEEE, 2008.

[8] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. Oreo: Detection of Clones in the Twilight Zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 354–365, New York, NY, USA, 2018. ACM.

[9] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chantal Roy, and Cristina V. Lopes. SourcererCC: Scaling Code Clone Detection to Big Code. In *International Conference on Software Engineering*. ACM, May 2016.

[10] Jeffrey Svajlenko and Chanchal K Roy. Bigcloneeval: A clone detection tool evaluation framework with BbigCloneBench. In *Proceedings of 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 596–600. IEEE, 2016.

[11] Jeffrey Svajlenko and Chanchal K Roy. Fast and flexible large-scale clone detection with cloneworks. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 27–30. IEEE Press, 2017.

Approved for Public Release; Distribution Unlimited.

[12] Jeffrey Svajlenko and Chanchal Kumar Roy. Evaluating clone detection tools with bigclonebench. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*, ICSME '15, pages 131–140, 2015.

[13] Di Yang, Aftab Hussain, and Cristina Videira Lopes. From query to usable code: An analysis of stack overflow code snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR'16, pages 391–402, New York, NY, USA, 2016. ACM.

[14] Di Yang, Pedro Martins, Vaibhav Saini, and Cristina Lopes. Stack overflow in github: Any snippets there? In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 280–290, Piscataway, NJ, USA, 2017. IEEE Press.

Approved for Public Release; Distribution Unlimited.

ACRONYMS

FQN	Fully Qualified Name
JAR	Java ARchive
JBF	Java Build Framework
NPM	Node Package Manager
SLOC	Source Lines of Code
SO	Stack Overflow
URL	Uniform Resource Locator