Some title

Author1
University of California, Irvine
a1@uci.edu

Author2
University of California, Irvine
a2@uci.edu

Author3
University of California, Irvine
a3@uci.edu

***Abstract*—This is an abstract.**

I. Introduction

For statistical works size is a relevant dimensions, thus software analysis relies more and more on large datasets of software. Relying on large datasets of software such as BOA, Sourcerer or the entire infrastructure created for the MUSE program is good because it reduces risks of under-representativeness of some niches.

As we have seen in out [accepted] SCAM paper, there are risks inherent to large datasets, but statistically size is relevant and despite the recognized concerns, an observation derived from thousands of units is generally better than one derived from a few dozens.

Software analysis poses a special risk for analysis, that grows with the size of the dataset. A statistical work on human population or traffic has the advantage of a guaranteed assurance the units are completely independent. The biological processes of persons or the mechanical and electronic constituents of automobiles do not intersect in any single way. The respiratory system of a person is completely free of influences from another respiratory system, same is true for a car's brake systems, etc. For software this is not the case.

Dependencies is an obvious case. Software is relying more and more on general software and frameworks. Apache commons is used in thousand of software projects, Oracle's SDK is universally used by any Java project, Google's JQuery by most websites. Most modern operative systems have advanced systems just to manage dependencies, such as yum or apt-get on LINUX or homebrew for Apple OS's. Most languages have central repositories and tools to manage dependencies and allow easy integration of artifacts such as hackage for Haskell, Maven for Java, pypi for Python, rubygems for ruby, etc.

Another important phenomenon of intra-programs similarities is code cloning. Either through copy-paste of small pieces of code or through the purposed addition of source code files form different origins to one's project, different programs can possibly share the same source code.

This is a studied phenomenon [many references from works from Mondego group], and programs are known to have pieces of code from online sources such as stack overflow. We know cases of methods duplication in Java, function duplication in Python, etc.

This phenomenon has never been studied in large datasets of software, but has the potentially to impact current software analysis methodologies and trends of using large datasets because code duplication will influence statistical analysis, and pieces of code or source files that are highly cloned will bias results towards their own characteristics.

Thus, we expect to answer the following question: How does the code cloning phenomenon affect large datasets of software, through which processes does it appear, and more important can it impact large-scale software analysis works?

In order to answer this question we analyzed the existence of code cloning on projects obtained through the well-known website GitHub, through the analysis of projects from 4 different programming languages.

Important to say that scripts, SourcererCC and lists of file-level and project-level cloning are publicly available.

II. Datasets and Methods

In this section we explain datasets, techniques and algorithms together with the set and order of directives used to present the results of this work; the order of the sections represents the order of the work flow we followed. The software used can be found at https://github.com/Mondego/SourcererCC; the version used was the one of commit COMMIT NUMBER OR DATE. Whenever possible, the software used was kept deliberately small and simple for understandability and replicability.

Scripts must be put in github.

A. Datasets

All the source code in Java and C/C++ was extracted in the context of the project Mining and Understanding Software Enclaves (MUSE)[1], by the Defense Advanced Research Projects Agency (DARPA)[2]. Software projects were obtained on a straightforward and simple process; they were gathered using direct searches through GitHub API[3], and afterwards crawled and downloaded.

For each repository, information related to branch name and last commit were maintained; forked repositories were filtered (to avoid biasing duplication values) and all the projects associated with a specific language were are defined in GitHub as having in majority source code in that language[4].

In Table I we can see basic information regarding the size of the datasets.

[1]http://www.darpa.mil/program/mining-and-understanding-software-enclaves
[2]http://www.darpa.mil
[3]https://developer.github.com
[4]https://github.com/blog/1037-highlighting-repository-languages

TABLE I: Dataset statistics.

[redacted]	Java	C/C++	JavaScript
# projects	54,627	62,226	
# files	5,856,349	34,478,611	

B. File-hash

The first step of clone detection was to calculate the hash value of all the files in a certain language, group them by equal hash, and create a list of files representing unique, distinct hashes. The rationale is that two files that have the exact same hash value (are hash-equal) are necessarily 100% clones, and there is no need to input these into a clone detection tool. These hashes were calculated on the entire file, on it's original form.

To calculate the hash of the files we used Python's hashlib (version 2.5) which "implements a common interface to many different secure hash and message digest algorithms"[5]. In particular we used the MD5 hash function, which creates a 128-bit hash that we later converted into a hexadecimal form (to avoid uncommon characters interfering with the intermediate textual data we generated).

C. Tokenization

All the analyzed source code went through a pre-processing phase where every source code file was transformed into it's tokenized form. The tokenization is the process of transforming a file into a 'bag of words', where each individual string (token, from now on) on the file is isolated and grouped by frequency. Please consider the small Java program below:

Listing 1: Java Foo.

```java
package foo;
public class Foo {

  // Example Class for ICSE paper

  private int x;

  public Foo(int x)
  { this.x = x; }

  private void print() {
    System.out.println("Number: " + x)
  }

  public static void main() {
    new FooNumber(4).print();
  }
}
```

The tokenization of this program will first remove comments and terminal symbols from the language, and then group the resulting tokens by their frequency, generating the following result:

```
Java Foo:[(package,1),(foo,1),(public,3),
(class,1),(Foo,2),(private,2),(int,2),(x,5),
(this,1),(void,2),(print,2),(System,1),
(out,1),(println,1),(Number,1),(static,1),
(main,1),(new,1),(FooNumber,1),(4,1)]
```

[5]https://docs.python.org/2/library/hashlib.html

where the tokens 'package' and 'foo' appear once, the token 'public' appears three times, and so on. Note that order is not important on the tokenized form.

D. Token-Hash

The list of the tokenized files is all the input the clone detection tool (SourcererCC) needs, but since this is the most time-consuming step in the pipeline we had a prior optimization step.

Specifically, this step removes all the files whose tokenized form is the exact same, avoiding clone analysis on two inputs that are the exact same.

Note that this step is different from the previous hash analysis: first, we calculated the hashes of all the files in their raw, original form; second, we calculated the hash of the tokenized form of the files to avoid duplication of input on clone analysis. Thus, note the difference between **file-hash**, the former; and **token-hash**, the latter.

E. SourcererCC

SourcererCC [**?**] is a clone detection tool developed with scalability as a main priority, and uses an heuristic-optimized index of tokens to significantly reduce the number of code-block comparisons needed to detect the clones, as well as the number of required token-comparisons needed to judge a potential clone. Testing has proved the tool to scale to a large repository with 250M lines of code in a single workstation (3Ghz i7 CPU, 12Gb RAM), and the values of precision to be 86% and recall 93%[6].

Explain in two or three paragraphs how SourcererCC works (token order by frequency, quick analysis of 20% of the tokens for a threshold if 80%, etc)

After tokenization we discard all the files that contain less than 65 tokens. This process was required since files that are too small can not be analyzed for clone detection with a reasonable degree of success. The tool was configured to detect file clones at 80% or more of similarity. No clone similarity is provided by the tool regarding two file clones; rather, the tool outputs a list of all file clone pairs, being all of these at 80% similarity or more. Remember this tool receives only the files that represent unique hashes, as explained in Section **??**.

NEXT SUBSECTIONS NEED TO BE ANALYZED

I don't think it's necessary here to explain how we created shards for SourcererCC. I give emphasis on hash isolation and union because the notion of 100% clones by hash will be an

F. Post-treatment of File Clones

The output of SourcererCC represents a list of file clone pairs at '80% or more' similarity, but this result was obtained only through the list of files representing unique hashes. We therefore need to add hash-equal files to complete the results. This required extending the results of SourcererCC in two steps:

1) All the files inside a group of hash equal files are file clones between themselves; therefore all the combinations

[6]Specific hardware details and testing conditions are in the original article. The value of recall varies under certain types of cloning.

of two files inside a group of hash-equal files represent a file-level clone pair.

2) We expanded the results from SourcererCC with the hash-equal groups of files, since for any two file clones A and B, all hash-equal files of A are also a clone of B and vice-versa. This can be formalized as: for any three files A, B and C:

$$if\ hash(C) = hash(A),$$

$$clones(A, B)\ \Rightarrow clones(C, B)$$

With this we have a complete list of file-level clone pairs, it is required only to obtain project-level clone information.

G. File-hash

The following step was to group all files by their respective project, and then proceed to find project-level clones. To do so, we search how many files in project A are clones of files in a project B. Similarly to the file-level cloning, only projects with 80% or more cloning were marked as project-level clones, but this time we provide the % of similarity. In particular, results are provided in the following format:

$$(A, B, \#\ files, \%\ cloning)$$

where:

- A is the project being cloned;
- B is the project that contains a clone of A;
- $\#\ files$ is the number of files in A that have clones in B;
- $\%\ cloning$ is the percentage of files in A that have clones in B. This percentage is directly related to the $\#\ files$ above.

In the end of this step we have all the data we need to start a statistical analysis of cloning.

In this section we will present the results obtained from analyzing the clones of both our subsets. In the next section we will provide a qualitative analysis of these results and discuss them accordingly. Since we are analyzing a repository of Java and a repository of C/C++ source code, for each presentation and discussion of a particular facet of the repositories we will present the results for these two programming languages side by side for comparison.

III. Results for File-level Clones

We start by providing the results for file-level clones, which abstracts files from the projects where they are contained and provides a repository-wide analysis on the behavior of cloning between source code files. This section is split between clones at 100% (exact replicas calculated through hash-equality, as describe in Section II), clones obtained using SourcererCC, which might not be exact replicas, and the total values for file cloning.

TABLE II: Empty files on C/C++ and Java.

	Java	C/C++
# empty files	3,153	55,779
% empty files	0.06%	0.52%
% hash-equal clones influenced by this files	2.8%	17.05%

TABLE III: Exact Replicas on C/C++ and Java.

	Java	C/C++
# files with at least one replica	2,688,886	9,196,287
% files with at least one replica	51%	85.07%
# clone pairs	24,167,643	553,223,258
ratio $^{clone\ pairs}/_{total\ \#\ files}$	4.61	51.17

A. Exact Replicas

Calculating exact replicas was performed by calculating an hash for the tokenized format of each file, and using it to group them. The results can be seen in Table III.

A first step on this process was to remove the empty files, i.e., the files whose tokenized form is empty (they might not be exactly empty, as the tokenizer dismisses comments); empty files would have necessarily the same hash and could interfere with further analysis. The values for empty files can be seen in Table II.

I kind of am on a dead-end here. We removed the empty files on Java, we did not on C, I am not sure how to justify this.

Source code files written in C/C++ show a considerably larger number of files with at least one exact replica. Although Java has a substantial amount of cloning with more than half the files having an exact clone (51%), the value for C/C++ source files is very substantial, with a large majority of files being in this group (85%).

The values for the percentage of files with exact replicas ends up reflecting on the ratio between the number of clone pairs and the total number of files. What this ratio tell is that for Java, each file is expected to appear, on average, on 5 (4.6, to be precise) file-level clone pairs, while this number for file written in C/C++ is of 51. This suggests that files with exact clones are very common in general, extremely common on C/C++.

B. Partial Clones

We follow now for the analysis of the amount of cloning between files that have small differences between them, namely, that are clones at '80% or more'.

It is worth mentioning that although we analyzed perfect replicas in the previous section, it is possible some of the file clones presented here are exact replicas as well. The reason being that hash equality is extremely sensitive to the contents of the file; two files can be exact replicas but the fact that one has an extra comment, or an extra new line ('\n' character) in the end will be enough for these not to be detected as hash-equals.

The results for partial clones, as produced by SourcererCC, can be seen on Table IV.

TABLE IV: Partial Clones on C/C++ and Java.

	Java	C/C++
# files with at least one clone	1,912,142	5,632,200
% files with at least one clone	36.49%	52.1%
# clone pairs	63,028,627	14,160,543
ratio $^{clone\ pairs}/_{total\ \#\ files}$	12.03	1.31

TABLE V: Total File-level Clones on C/C++ and Java.

	Java	C/C++
# files with at least one clone	3,286,697	9,561,223
% files with at least one clone	62.72%	88.44%
# clone pairs	87,196,270	567,383,801
ratio $^{clone\ pairs}/_{total\ \#\ files}$	16.64	52.48

For Java, the percentage of file cloning is still substantially smaller than C/C++, with a difference of around 15%. On the other side, Java has a much larger ratio of clones pairs per file, with each file being expected to appear in 12 clone pairs. The disparity between a relative smaller number of files being clones and a relative large ratio of files per clone pair suggests that in Java each file is on average cloned a substantial number of times, therefore leveraging the ratio.

For C/C++ we have the opposite result, with half the files having a file clone but a extremely small ratio of 1. We still see that by comparison C/C++ has a larger number of file cloning, but this data suggests that these types of file clones are isolated cases, and while it is true it seems files are reused and adapted, this phenomenon seems to be casual since the indication is that it is very uncommon for the same file to be reused twice, contrary to what we have seen for Java.

C. Total Results

From analyzing the two previous sections, it seems rather misleading how for C/C++ files, there is a substantial files with exact replicas, of more than 50%, while at the same time the same is true for partial clones. The explanation for this phenomenon will be provided in this section, where we analyze the total number of file clones for the two repositories. The results can be seen in Table V.

The total number of file clones indicates a large number of file clones, and shows the phenomenon of file duplication is wide spread at least on the scopes of programming languages we are analyzing. Even the relative small amount of file cloning on Java at 63% shows that more than half the files have a file clone, while the proportionally the observation C/C++ results shows only a very small amount of files of around 12% does not have a similar file in the repository.

The ratio between the number of file clone pairs and the number of files is revealing of another phenomenon: not only files are widely cloned, as we have see above, they are also cloned a substantial number of times. At 16 file clone pairs per file, Java shows a considerable re-usage of files. At 52 file clone pairs per file, source code files in C/C++ seem to be quite reused. How this phenomenon appears and how do files seem to propagate along the repository is something for which a set of possible explanation will be provided in Section **??**.

TABLE VI: Total Project-level Clones on C/C++ and Java.

	Java	C/C++
# project clone pairs	22,381	17,135
# projects containing 1 or more clones	4,200	5,021
% projects containing 1 or more clones	10.64%	9.85%
# projects being cloned	3,315	3,802
% projects being cloned	8.40%	7.46%

Bringing the results of exact replicas and partial clones to the total number of file-level cloning, we can see that for C/C++ only 3% of the files that have a clone do not have an exact replica. and that around 36% of the files that have a clone do not have a partial clone. Resuming, exact replicas are wide spread on the repository of C/C++ source code files.

For Java exact replicas are also a substantial part of file cloning. Of all the files that have a clone only around 10% do not have an exact replica and around half has no partial clone. Proportionally the amount of file cloning and the amount of exact replicas is smaller than what was found for C/C++ suggestion variation between programming languages.

As far as these two repositories suggest, file cloning (and exact replicas in particular) seem to be a wide spread phenomenon in source code, although the reasons are still unclear. On the next section we will analyze project-level cloning on a similar fashion as we have done for file-level.

IV. Results for Project-level Clones

Having analyzed file-level clones for Java, we focus our attention on project-level clones. As stated in Section II, at file-level directionality is not important, as a clone pair indicates two files that are mutual clones. On the other way, at project-level directionality of project clone pairs matters, as a project pair presented as $(P_1,\ P_2)$ means that a clone of P_1 is inside the contents of P_2, being the opposite nor necessarily true; a simple example would be a small dependency P_1 existing inside a large software artifact P_2. For this reason, we shall look at project cloning from two perspectives: the phenomenon of projects being cloned inside other, possibly larger projects, and the phenomenon of a project containing various, possibly smaller projects inside it's contents.

The results for project-level cloning can be seen in Table VI, split between projects containing clones, and projects being cloned.

The first interesting result is that the ratio of clone pairs as comparing to the total number of projects (recall that for Java we are analyzing 50,979 projects, and for C/C++ 39,584) is considerably smaller than the ratio we found for file-level clone analysis.

It is worth considering why this ratio, and the overall percentages of cloning, have such a wide gap from file to project-level (for C/C++ for example we had 88.44% of files cloned). The explanation is relatively simple: for file level we are analyzing every single file, but for project-level we are only interested on the file clones that are contained within projects which are clones at 80% or more. This means there is a large slice of files which would flag projects as clones at a lower similarity, let us say 20 or 30%, maybe less, which we are simply discarding.

Returning to the results of of Table VI, it is interesting to notice that the values are very similar between the two repositories. Independent of the causes behind the phenomenon of project cloning, just look at the relative values projects from Java and C/C++ seem to be cloned and possessing clones at a similar rate, with a small difference of 0.96% for the former, and 0.79% for the latter.

It is also noticeable that for both repositories more projects contain clones than projects are cloned, as by randomly picking a projects it is more likely it has clones inside than it is that he is cloned somewhere else. It is important to note that this relative values in percentage do not totally explain the phenomenon of project cloning, as saying that 8.40% of C/C++ projects are cloned somewhere does not provide any information regarding how many times on average are projects cloned, only that 8.40% projects are cloned 'at least once'. This also justifies why the relative number of projects that contain clones is larger than the relative number of projects that are cloned.

In order to clarify cloning frequency as present in Table VII the distributions of all of the four components on project-level clone pairs (recall, from Section II, that project level pairs are presented in the format $(A, B, \#\ files, \%\ cloning)$). Results are provided for C/C++ on top, and Java on the bottom.

A. *Distribution of Percentage of Cloning and Number of Files Involved*

We shall start analyzing the percentage of cloning. We see that the mean is slightly larger on JAVA (by around 5%), with an equally similar standard deviation. There are relatively more clones at 100% on Java, given that the 75% quartile is 100% for this language but 93.66% for C/C++, but by comparison, we would say the similarity of project-level clones for Java and C/C++ is similar. The minimum and maximum values are an obvious 80% and 100%.

Next we look at the number of files involved in cloning, presented on the second column. Here values are substantially different for Java and C/C++, with the values of the mean and standard deviation to be pretty much double when comparing both repositories. The values at the 75% quartile are an order of magnitude larger on C/C++.

Since there is a direct relation between between the first column with the percentage of cloning and the second column, with the number of files involved in such cloning, but the value of percentage of cloning are similar between the two repositories, and the values for the number of files are substantially different; the explanation is that on average C/C++ projects are larger, meaning that between two projects-level clone pairs with the same percentage of cloning it is expected for the C/C++ clone pair to have more files involved. This is something we had already concluded by looking at the Table I with the statistics of the datasets, where it clear that for a smaller number of projects there is a substantially larger number of files for C/C++.

A final note on th discussion on the number of files involved in project clone pairs, the values of the maximum number of files found for Java and C/C++ is expectedly larger for the latter, for reasons explained above.

B. *Distribution of Projects being Cloned*

We now focus the discussion on the distribution of the number of projects being cloned, presented in the third column. As we have seen in Table VI the percentages are of 8.4% for Java and 7.46% for C/C++.

Looking at Table VII a first observation is that a large number of projects being cloned is cloned only once. This is more true in Java than it is in C/C++, since the 50% quartile is 1 in the former but 3 in the latter.

Looking at the values of mean and standard deviation it is also observable that projects are on average cloned more times on C/C++ (mean of 6.75, standard deviation of 10.5) than they are on Java (mean of 4.5, standard deviation of 8.59), corroborated by the fact the 75% quartile gives the indication a quarter of the projects being cloned are being so 5 times on Java but 8 times on C/C++.

A graphical comparison of distribution of projects being cloned can be seen in Figure 1. It is clear the number of times projects are cloned grows both sooner and faster for C/C++ (represented by the dotted, green line).

The maximum values for projects being cloned is larger on Java than it is on C/C++ (277 vs 141). We consider these outliers that do not relate to the trend presented by the distribution (the reason why the char of Figure 1are artificially limited on the y axis). Nevertheless it is interesting to look at which projects represent these outliers.

For Java the project being cloned more times is `xianminvip/androidrss`[7]. This projects is constituted by a single, automatically generated file, which supports the theory of these projects being a uninteresting outlier. The reason this projects appears cloned so many times is due to the fact that it has only only, possibly very common file (since it is automatically generated), meaning that for every single project that contains the same file this project `xianminvip/androidrss` is flagged as a clone at 100%. For C/C++ the project being cloned more time is ...

V. Discussion

On file-level, C has much more hash-equal clones, Java more partial clones. THis suggests different ways of handling these two, for example, space-management on github would be much easier for C, as hash-equality would do a good job, but much harder for Java.

VI. Conclusion and Future Work

References

[1] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big code," *CoRR*, vol. abs/1512.06448, 2015. [Online]. Available: http://arxiv.org/abs/1512.06448

[7]https://github.com/xianminvip/androidrss/tree/admin

TABLE VII: Distribution of Values for Project Cloning.

		JAVA		
	% cloning	**# files involved in cloning**	**# projs. being cloned**	**# projs. with clones**
mean	93.93%	556.48	4.5	3.41
std deviation	6.8%	1,645.81	8.59	5.03
min	80%	1	1	1
25% quartile	87.8%	6	1	1
50% quartile	97.4%	93	1	1
75% quartile	100%	202	5	3
max	100%	31,876	277	97
		C/C++		
	% cloning	**# files involved in cloning**	**# projs. being cloned**	**# projs. with clones**
mean	89.36%	1,648.74	6.75	5.32
std deviation	6.49%	2,988.71	10.5	7.09
min	80%	1	1	1
25% quartile	83%	25	1	1
50% quartile	88%	134	3	2
75% quartile	93.66%	1322	8	6
max	100%	107,694	141	44

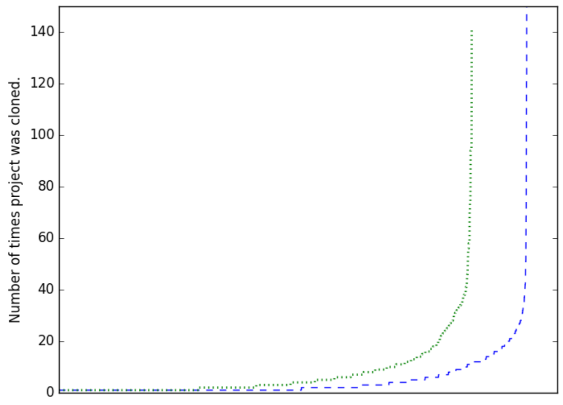

Fig. 1: Distribution of the number of times projects are cloned. The dotted, green line represents C/C++; the dashed, blue line represents JAVA. Charts are artificially limited on the y-axis.

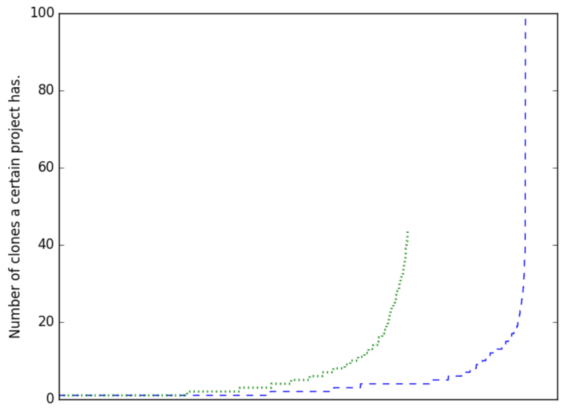

Fig. 2: Distribution of the number of clones per project. The dotted, green line represents C/C++; the dashed, blue line represents JAVA. Charts are artificially limited on the y-axis.