ACAnon : Automatic Code Anonymization to Enable Code Sharing

Aashish Karki
Amherst College
akarki15@amherst.edu

ABSTRACT

A developer might choose against sharing code on a developer help forum like StackOverflow because the code snippet may contain a proprietary algorithm or her original idea. I built and tested a code anonymization technique called ACAnon, that finds an open-source code snippet similar to the original one. The developer can then share the similar code on the help forums.

1. RESEARCH PROBLEM AND MOTIVATION

Developer help forums like StackOverflow[1], CodeProject[2], and Google Groups[3] are popular among software developers for troubleshooting code. However, a developer may decide against sharing a problematic code snippet because it contains *sensitive information*. In the context of this paper, sensitive information means method calls, literal values, and method structures that may divulge details about proprietary algorithms or an original idea.

Let us take an example of a hypothetical software developer Mary. While working on her project, she gets stuck on the method `readFile()` shown in Code 1. She gets a `FileNotFoundException` on line 2. Her preferred option is to share it on StackOverflow. Sharing the code, however, gives away sensitive information like method and variable names. One option could be manually replacing the sensitive information with placeholders. In this paper, I refer to this process of removing sensitive information as code anonymization. Since manual code anonymization does not guarantee that all sensitive information is replaced, she decides against sharing the code altogether and ends up spending hours debugging the code. It would be convenient if she could automatically anonymize the code snippet and post it on StackOverflow for quicker feedback.

[1]`http://stackoverflow.com`
[2]`http://www.codeproject.com`
[3]`https://groups.google.com`

Code 1 : Hypothetical code snippet containing sensitive information

```
1 void readFile(String s) throws IOException{
2       BufferedReader bReader = new BufferedReader(new
            FileReader(s));
3       String string;
4       while ((s=bReader.readLine())!=null){
5         System.out.println(s);
6       }
7   }
```

In this paper, I propose a novel way called ACAnon (**A**utomatic **C**ode **Anon**ymization) that replaces an original code snippet with a *similar* open-source code snippet. It is, however, difficult to unambiguously define similarity as it may vary on a case-to-case basis. For code anonymization, I define that two code snippets are similar if they have the same Abstract Syntax Tree (AST) representation.

An AST maps Java source code to a tree format with nodes as leaves of the tree [2]. Using AST as a basis of comparison allows ACAnon to compare code snippets line by line. ACAnon suggests the open-source[4] code snippet, Code 2 as a replacement to Mary's code snippet depicted in Code 1. Now, Mary can share Code 2 without worrying about disclosing sensitive information.

Code 2 : Open-source code snippet similar to Code 1

```
1 static void copy(String filename, PrintStream out)
      throws IOException {
2       BufferedReader br = new BufferedReader(new
            FileReader(filename));
3       String s;
4       while ((s = br.readLine()) != null) {
5          out.println(s);
6       }
7   }
```

My contribution is an AST similarity-based technique, ACAnon[5], that automatically anonymizes a code snippet by replacing it with a similar open-source one.

2. BACKGROUND AND RELATED WORK

ACAnon draws inspiration from previous works on *code obfuscation* [3]. Code obfuscation changes a software's operation, data, organization, and flows while retaining

[4]`http://tinyurl.com/ppgqlwy`
[5]`https://bitbucket.org/akarki15/projectchooser`

computational performance and accuracy [3].

Ertaul and Venkatesh [1] implement several code-obfuscating algorithms in *JHide*. JHide uses code-obfuscating algorithms to increase the time and effort required to reverse engineer a Java class file. The algorithms decrease the class file's human readability. On the other hand, ACAnon anonymizes code without decreasing its human readability. JHide processes Java class files while ACAnon processes Java source files.

3. APPROACH AND UNIQUENESS

3.1 Implementation

ACAnon starts by downloading GitHub's top 100 open source Java projects. It then creates an AST for each method body. Processing each method body is easier since methods in Java are not nested and do not overlap. It uses the method body's AST to generate an *Intermediate Representation* (IR) for that method body. I define the IR for a method body as an in-order traversal of the method body's AST nodes. ACAnon uses these IRs to compare two method bodies. Two method bodies are similar if their IRs are exactly the same (both in size and order). ACAnon creates a library of IRs for all the methods.

For example, Code 3 is a part of the IR generated for both Code 1 and 2. ACAnon thus considers them to be similar.

Code 3 : Intermediate representation for Code 1 and Code 2

```
1 "Block;PrimitiveType;SimpleName;SingleVariableDeclaratio
2 n;"... ..."SimpleName;SimpleName;"
```

3.2 Analysis Technique

I analyzed the library of IRs to test the frequency of open-source snippets suggested by ACAnon. To measure the length of a method body, I used the number of AST nodes in it as opposed to lines of code. Using the number of AST nodes over lines of codes gives a better estimate because it takes into account method bodies with long, one-line statements and nested code-blocks. Accurate measurement of method body length is important for analysis.

To measure how frequently ACAnon suggests an open-source code snippet, I introduce the concept of a *hit*. A method m is said to have a *hit* if there is another method n that produces the same IR as produced by method m. Using the IR library, I check whether each method in the downloaded GitHub projects had a hit. I use this information to calculate the percentage of hits (%hit) for method bodies of various lengths.

4. RESULT AND CONCLUSION

Figure 1 plots the average %hit for a *bucket*. A *bucket* numbered n contains method bodies whose sizes range from $10n - 9$ to $10n$ AST nodes. For example, bucket 1 contains method bodies whose sizes range from 1 to 10 AST nodes.

The result indicates that the average %hit for a bucket decreases exponentially as methods in the bucket contains more AST nodes. Shorter method bodies have a higher %hit. For example, method bodies in Bucket 1 have a high average %hit of 93.7%. This is expected because this bucket mainly contains method bodies with empty content (dummy methods), or simple one-line statements like `return _partial;`, `return _staticCtor != null;`, or `super(name);`.

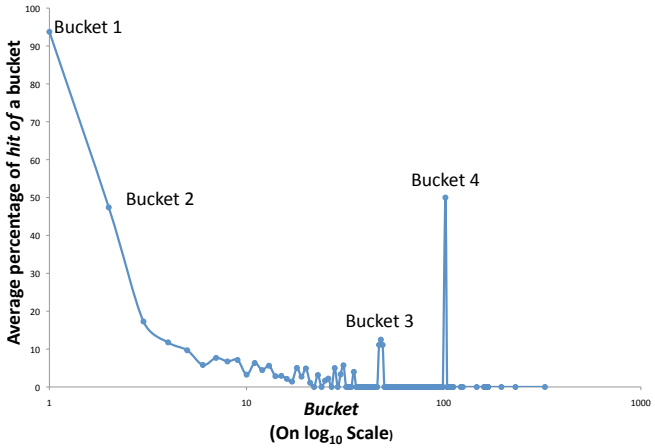

Figure 1: *Bucket* vs. Average percentage of *hit* of a bucket

As methods get longer and more complex, the average %hits drastically decreases. For example, Bucket 2 on the graph contains methods with 11 to 20 AST nodes, and has an average %hit of 47.39%. This is because Bucket 2 contains more varying types of method bodies than Bucket 1. Bucket 2 contains methods with simple multi-line statements as well as methods with single *complex* statements meaning more AST nodes like `mActionBar.setNavigationMode(mode);`, or `return extension.equals(what);`.

Buckets 3 and 4 are two notable exceptions to the exponential decrease in the graph. These exceptions are due to entire method bodies being copied between projects. For example, Bucket 4 contains only 3 methods, 2 of which were duplicate methods from two different projects. Bucket 3 also has several cases of cross-project duplication.

Analysis of open-source suggestions given by ACAnon shows that as method bodies get longer, it is harder to find an open-source substitute for them. To address this, future works can relax the definition of *similar* code-snippets, increasing %hits for method bodies of all lengths. Doing so would help find open-source code snippets for long methods.

5. REFERENCES

[1] L. Ertaul, S. Venkatesh, JHide - A Tool Kit for Code Obfuscation, *IASTED Conf. on Software Engineering and Applications, page 133-138. IASTED/ACTA Press,* 2004

[2] Kuhn, Thomas, and Olivier Thomann, Abstract syntax tree. *Eclipse Corner Articles 20,* 2006

[3] M. R Stytz, J. A Whittaker, Software Protection - Security's Last Stand, *IEEE Security and Privacy,* January/February 2003