Source Code Plagiarism Detection with Abstract Syntax Tree Fingerprinting

Kruthika K

Abstract

Source code is the bedrock of our Internet age. At academic and corporate institutions, the legal and disciplinary ramifications of plagiarism are extremely serious. At the same time, detecting plagiarism is tedious and error-prone with the human eye alone. Many systems accomplish plagiarism detection but are expensive to compute at scale.

In this paper, we survey state-of-the-art architectures for source code plagiarism detection. We implement Stanford's MOSS architecture and build upon it with Abstract Syntax Tree Fingerprinting. Further, we survey set similarity metrics and propose our own to account for sequence matching in files.

We report performance of several document fingerprinting methods and similarity measures on a custom corpus of 100+ source code files. We conclude that MOSS is a gold standard in accuracy and scalability, but fingerprinting an AST with a Merkle-like AST hashing technique can improve precision.

1 Introduction

Source code is easy to duplicate but duplication is difficult to detect. In an academic or corporate context, intellectual property is taken especially seriously. Detecting exact copying is easy enough, e.g. document checksums, but if the plagiarizer is smart and changes the copied code in any way, detection is considerably more difficult.

String based techniques computing edit distance or similar metrics are effective but very slow in practice. In a class of one hundred students, for example, comparing 100^2 code pairs may take hours if not days. Speed and scalability must be preserved.

Thus, we have several goals for a plagiarism detection model. In summary:

1. *Scalability*: The model should take a reasonable amount of time to compute for larger input files and file sets
2. *Noise insensitivity*: Miscellaneous features of a source code file should not be taken into account. This includes white space, common language features and idioms, etc.
3. *Tolerance*: The model should be robust against attacks involving changing original source code to look different while maintaining correctness (i.e. renaming variables/methods, rearranging code blocks, adding extraneous and redundant features (e.g. `if(true){...}`).
4. *Security*: The model should be able to conceal sensitive information in source code.
5. *Flexibility*: The model should not rely on the size of the document set for precision, maintaining consistency between large and small document sets.

Many methods separate treating source code as raw text versus as a program. The technique proposed in Aiken et al. [1] encodes source code as a sequence of hashed k-grams and extracts a characteristic set of fingerprints. This is a very scalable approach, however treating source code as raw text is more difficult to preprocess to remove useless and redundant language features. It also may not take full advantage of the structure and features of a modern coding language. Treating code as a more complicated structure like an AST (Abstract Syntax Tree) or PDG (Program Dependence Graph) or similar intermediate representations, while permitting finer grain analysis of a program, can quickly become expensive to handle when computing similarity. Surely there is a trade-off.

Indeed, if we think of plagiarism detection as a number of moving parts, there is room for much experimentation. In this paper, we implement the MOSS model [1] using the state-of-the-art k-gram winnowing technique. We experiment with multiple AST encoding methods in order to fingerprint an AST. We also compare a number of similarity measures for computing set similarity and propose a novel metric. Finally, we evaluate performance on a custom corpus of 100+ documents and find that certain AST encoding methods can improve precision.

2 Related Work

Many known methods of plagiarism detection are able to satisfy goals 2,3 and 4, but preserving scalability and flexibility is quite challenging for exhaustive string matching. Many AST comparison methods [2,6] use clustering algorithms to find similar document pairs, however these algorithms are difficult to implement and evaluate, and can only detect similarity in large sets of documents.

2.1 Fingerprinting Source Code

To satisfy the first goal, a document must be significantly reduced in size while still preserving enough information to maintain an accurate similarity analysis. Document fingerprinting algorithms are suited for such a task. As previously mentioned, the winnowing algorithm proposed in Aiken et al. is implemented in an online tool for source code plagiarism detection called MOSS (Measure of Software Similarity) [1]. We can phrase the MOSS system as three distinct parts: encoding, fingerprinting, and similarity scoring.

In the encoding step, a source code file is stripped of extraneous features (e.g. whitespace, irrelevant languages features, etc.) and then shingled into a sequence of k-grams. Each k-gram is then hashed (their paper does not specify how so we use a cryptographic hash function).

In the fingerprinting step, a subset of hashed k-grams is chosen as our characteristic set S. This is done with the winnowing algorithm which extracts each fingerprint in a rolling window over the hashed k-gram sequence while preserving file position information. If density d is the fraction of fingerprints selected, S will have an expected density of $d = \frac{2}{w+1}$ for a rolling window size of w [1].

In the similarity scoring step, two characteristic sets are compared with a similarity measure like Jaccard or Cosine similarity [5]. The specific similarity measure in the MOSS implementation is not discussed so we implement several — Jaccard, Binary Cosine, and Sorensen-Dice Coefficient.

Because this pipeline is fairly modular, there is a great degree of room for experimentation in preprocessing and similarity computation. This also aligns with our security goal: assuming some collision resistant cryptographic hash function is used, it is practically impossible to determine the contents of the original file from a set of fingerprints. Since we only need to store the characteristic set of fingerprints to compute similarity, we can easily design a system that is blind to source code content while accurate in similarity detection.

2.2 Source Code as an AST

MOSS only works if we treat source code as raw text, but it is possible to encode source code differently (given that it is syntactically correct and compiles). The most promising alternative to Aiken et al. is to represent source code as an Abstract Syntax Tree. Many AST based methods either compare the trees directly (this is naive) or find some way to encode the tree into a characteristic set and then perform comparisons.

The *DECKARD* system uses an AST to generate a set of characteristic vectors for each file that can then be clustered with locality sensitive hashing to find the most similar documents [6]. Chilowicz et al. use several methods to fingerprint an AST, the fingerprints are then stored in a database and can be clustered to compute any document collisions [2]. Sequences of similar AST subtrees can then be extracted with a suffix array [2].

For the purposes of this paper, we will not focus on these clustering methods, rather we are concerned with the task of computing the similarity of two documents, per goal 5. If a performance comparison is necessary between the two system types, note that many clustering methods must store information greater than the size of the file (for everything in the file AND in the AST) while the MOSS method significantly reduces memory needed per document.

It is also of note that representing source code as an AST and encoding it as a sequence of fingerprints fits nicely in the MOSS pipeline. In addition to our MOSS evaluation, we implement two AST fingerprinting methods from Chilowicz et al. into the encoding step of the MOSS pipeline and evaluate their performance.

3 Methods

3.1 Robust Winnowing

As previously summarized, winnowing will extract a characteristic set of fingerprints from a sequences of n hashes $h_1 h_2 \ldots h_n$. Using a rolling window of size w, we will select the minimum right-most hash from each window.

This algorithm outperforms similar algorithms using a local, context sensitive approach (hashes are selected relative to their window, not some external measure) for fingerprint selection. It has been shown that the winnowing algorithm performs within 33% of the lower bound and is very accurate on real world data [1].

As suggested in their paper, we use a k of 5 and a w of 5.

3.2 AST Fingerprinting

We examine three AST Fingerprinting techniques proposed in Chilowicz et al. The hash of a node N is implemented as using a cryptographic hash function on the string representation of that node, including type, variable name, and other annotations. For our experiments, we used the MD5 hash function. This may pose security flaws (MD5 has been broken) and a performance difference in a production system, but for our purposes there is no significant impact.

Merkle-like

A Merkle tree is a binary tree where each node is the hash of the concatenation of the hashes of its children [3]. Merkle trees have the property that if any one node's value is changed, the hash of the root must also change assuming some (practically) collision resistant hash function. Thus each node in the tree has a unique hash, and if two nodes share a hash they must have the same contents.

An AST is not always a binary tree, but the same logic may be applied. The Merkle-like fingerprinting method represents each node as the concatenation of the hashes of its children and its own value. The tree is traversed depth-first and the resulting hash sequence is returned, ready to be winnowed.

Node hashing

We might also hash each node independently, taking each node without sub tree information and use the resulting preorder traversal sequence.

Dyck Word with Karp-Rabin Hashing

Chilowicz et al. [2] present a novel fingerprinting method which represents each AST node as a Dyck Word. A Dyck Word $\mathscr{D}(\alpha)$ is simply a tree traversal represented by the recurrence relation:

$$\mathscr{D}(\alpha) = \begin{cases} \text{if } \alpha \text{ is a leaf} & V(\alpha)\cdot V(\alpha) \\ \text{if } \alpha \text{ is not a leaf} & V(\alpha)\cdot \mathscr{D}(c_1)\ldots\mathscr{D}(c_n)\cdot V(\alpha) \end{cases}$$

Where α is a node, V is some hash function, and $c_1,\ldots,c_n$ are children of α.

We can then use the Karp-Rabin [4] hash function $\mathscr{K}$ to hash each node:

$$\mathscr{K}(a_1a_2\ldots a_m) = ((a_1\times B + a_2)\times B + a_3)\ldots)\times B + a_m$$

for some prime number B and Dyck word $a_1a_2\ldots a_m$. This hashing method is a rolling hashing method which is cheaper to compute.

3.3 Similarity

There are many methods of computing similarity between sets. We implement three measures: Jaccard Similarity, Sorensen-Dice Coefficient, and Binary Cosine Similarity. In addition, we propose a novel extension to the Binary Cosine metric.

Jaccard Similarity

Jaccard similarity between two sets X and Y is defined as the intersection over the union or

$$\frac{|X\cap Y|}{|X\cup Y|}$$

This allows us to measure the number of common items relative to the size of the entire set of fingerprints over both documents.

Sorensen-Dice Coefficient

This measure is similar to Jaccard similarity, though it does not obey the triangle inequality and thus cannot be used as a distance metric. The Sorensen-Dice Coefficient is defined as

$$\frac{2|X\cap Y|}{|X|+|Y|}$$

for sets X and Y.

Binary Cosine Similarity

For sets X and Y, consider the master vector M of size $|X\cup Y|$. M on its own is simply a zero vector.

We can encode sets X and Y using the master vector as M_X and M_Y respectively. $M_{X_i} = 1$ if the ith fingerprint of $X\cup Y$ is in X. Define Binary Cosine similarity as

$$\frac{M_X\cdot M_Y}{||M_X||\times||M_Y||} = \frac{M_X\cdot M_Y}{\sqrt{\sum_{i=0}^{|M_X|}M_X[i]^2}\sqrt{\sum_{j=0}^{|M_Y|}M_Y[j]^2}}$$

Binary Cosine was chosen for its robustness against saturation attacks — the binary vector simply marks the presence of a fingerprint and not the amount of times it occurs.

Sequenced Binary Cosine Similarity

Consider the following dilemma: reducing a file to an unordered set of fingerprints in a vector space naturally loses positional information. Further, we might want to increase a similarity score if a sequence of fingerprints match in both files.

To do this we can extend our master vector M to incorporate all possible concatenations of each fingerprint. Let F be our set of fingerprints, f_1 and f_2 are the hash sequences for file 1 and file 2. We have a new master vector with size bounded above by

$$\sum_{i=0}^{m} n^i = n + n^2 + \ldots + n^m = \frac{1-n^m}{1-n}$$

where $n = |F|$ and $m = \min(|f_1|, |f_2|)$. The theory behind this is we have a dimension in our vector for ever possible sequence of fingerprints bounded by the size of the smaller file. The vectors M_{f_1} and M_{f_2} become very large very quickly, but are naturally very sparse. Thus there is an efficient way to compute their cosine similarity.

For the dot product, we compute the size of the intersection between f_1 and f_2. For every fingerprint i shared in common, we maintain two sets r_1 and r_2 of all the fingerprints directly following i in f_1 and f_2. For every fingerprint in those sets we increment the dot product by $|r_1|$, reduce the sets then repeat until both r_1 and r_2 are empty. In the worst case, both files are the same in which case we will traverse the entire f_1 and f_2 for every fingerprint shared in common (which is all of them). Thus this algorithm is bounded above by $O(|f_1|^2 + |f_2|^2)$. We suspect this might be sped up to linear time with dynamic programming algorithms.

The product of the magnitude of the vectors can be computed with a nested loop of a rolling window which gradually increases in size from 1 to $|f_z|$ for file z. On each rolling window, we track the sequences we've already seen as to not double count. This is expensive to compute (quadratic) and in practice

$$\sqrt{\sum_{i=1}^{|f_z|} i} = \sqrt{1+2+\ldots+|f_z|} = \sqrt{\frac{|f_z|(|f_z|+1)}{2}}$$

is a good approximation and upper bound.

An unfortunate trade-off for accuracy is time and space: we need to store the original hash sequence of each file instead of a characteristic set. This hurts scalability but is still $O(n^2 + m^2)$ — quadratic in the size of the two files which is still quite fast compared to other methods.

In practice, this similarity is very sensitive, often under reporting similarities. However, it is very good at distinguishing plagiarism from non-plagiarism. Typically non-plagiarized file pairs score very low (0 - 0.03) though plagiarized file pairs can score anywhere from 0.08 to 0.98. We apply the transformation

$$\max(0, 1 + c \log sim(f_1, f_2))$$

for some constant c and binary cosine function sim to compensate for the ultra-sensitivity and normalize the ranges (values will still be in $[0,1]$). We've found a c of $\frac{1}{7}$ yields the best results. A mathematically rigorous justification has yet to be conceived, and for the purposes of this paper we will simply observe its performance but not use it to draw any conclusions about fingerprinting techniques.

Thus our suggested metric SBC is:

$$SBC(f_1, f_2) = \max\left(0, 1 + \frac{1}{7}\log\frac{M_{f_1} \cdot M_{f_2}}{\sqrt{\frac{|f_1|(|f_1|+1)}{2}}\sqrt{\frac{|f_2|(|f_2|+1)}{2}}}\right)$$

where M is the master vector of size $|\{f_1\} \cup \{f_2\}|$.

3.4 Known Attacks

Copy-paste plagiarism is simple to detect, but if the plagiarizer is smart, plagiarized code may be harder to detect with simple, subtle changes. Below is a survey of possible attacks that our model seeks to tolerate.

Name and Position Attack

An attacker may copy several code blocks and attempt to conceal the plagiarism by changing variable names and method names.

Saturation Attack

An attacker may note that if the plagiarized code is a relatively small portion of the entire assignment, the probability of detection is smaller. To take advantage of this, the attacker might fill the file with redundant code to "saturate" the characteristic set with useless information and reduce the probability of a collision.

Some similarity methods that take into account intersection proportional to size, e.g. Jaccard similarity, are susceptible to such an attack.

Redundancy Attack

If our encoding method encodes the structure of the AST and relies heavily on looking at similar AST structures to determine source code similarity, an attacker could simply add redundant features to the language, i.e. bury code under redundancy. For example, if we copy the function `invoke();`, we could do the following:

```
while(true){
    if(2 + 2 == 4) {
        invoke();
    }
    break;
}
```

which would not affect correctness but would certainly affect the structure of the AST.

Additionally, an attacker might rearrange the order of methods, variable declarations, etc. without affecting correctness.

4 Experiment and Results

We run several experiments on our model. We give a performance overview of each fingerprinting method using binary cosine as a baseline metric on a small test set of three files.

Fingerprinting Method	P1 vs P2	P1 vs P3	P2 vs P3
k-gram	0.78	0.23	0.23
AST-to-text	0.94	0.61	0.66
Merkle	0.61	0.19	0.17
Node	0.66	0.30	0.29
Dyck word	0.75	0.38	0.38
AST-to-text (no name)	0.95	0.76	0.74
Merkle (no name)	0.60	0.27	0.22
Node (no name)	0.78	0.62	0.60
Dyck word (no name)	0.75	0.38	0.38

Table 1: Comparing fingerprinting techniques with Binary Cosine

We then use the most promising fingerprinting methods to compare different similarity measures. We then run a comprehensive test on our source code corpus. In addition to the discussed AST fingerprinting methods from Chilowicz et al. [2], we use an additional method referred to as AST-to-text which is simply printing out the AST and feeding it into the k-gram MOSS encoder. Additionally, to test robustness against naming attacks we remove all variable names from ASTs and sort node children.

The three files used are three Java implementations of `PowerSet`, a function to print out the power set of an input string array. The first file P1 is plagiarized in P2 with changes to variable names and minor restructuring of code. The third file P3 is a separate implementation and should not be detected as plagiarism with either P1 or P2. These simple examples should show us which techniques are most effective in detecting plagiarism. We are looking for a large similarity score for P1 vs P2 and a low score for P1 vs P3 and P2 vs P3.

In Table 1, we see that the AST-to-text is very prone to false positives, especially when excluding the names of variables. Node hashing also flags P1 vs P3 and P2 vs P3 as plagiarism which is a false positive.

The more promising methods are the Merkle method and Dyck word method. Interestingly when we remove variable names from AST nodes, the scores stay within a one percent difference, which shows the fingerprints encode more about the structure of the program than the names. We can reduce our candidate fingerprinting methods to k-gram, Merkle, Merkle (no-name), Dyck word, and Dyck word (no name).

We run the same fingerprinting comparisons for Jaccard similarity, Sorensen-Dice coefficient, and Sequenced Binary Cosine in Tables 2-4.

4.1 Corpus

The corpus used consists of 100+ source code files. There are several sets of files which are constructed to be plagia-

Fingerprinting Method	P1 vs P2	P1 vs P3	P2 vs P3
k-gram	0.78	0.28	0.19
Merkle	0.63	0.19	0.18
Merkle (no name)	0.63	0.31	0.19
Dyck word	0.70	0.35	0.29
Dyck word (no name)	0.70	0.42	0.29

Table 2: Comparing fingerprinting techniques with Jaccard Similarity

Fingerprinting Method	P1 vs P2	P1 vs P3	P2 vs P3
k-gram	0.78	0.23	0.22
Merkle	0.61	0.20	0.17
Merkle (no name)	0.60	0.26	0.22
Dyck word	0.75	0.34	0.34
Dyck word (no name)	0.75	0.37	0.37

Table 3: Comparing fingerprinting techniques with Sorensen-Dice Coefficient

Fingerprinting Method	P1 vs P2	P1 vs P3	P2 vs P3
k-gram	0.61	0.28	0.26
Merkle	0.60	0.35	0.34
Merkle (no name)	0.64	0.36	0.36
Dyck word	0.64	0.46	0.46
Dyck word (no name)	0.64	0.50	0.50

Table 4: Comparing fingerprinting techniques with Sequenced Binary Cosine

	J	SD	BC	SBC
k-gram	0.84	0.87	0.89	0.91
Merkle	0.55	0.55	0.53	0.91
Merkle (no name)	0.61	0.94	0.92	0.78
Dyck	0.75	0.75	0.75	0.75
Dyck (no name)	0.76	0.75	0.75	0.80

Table 5: Precision of Fingerprint Methods vs Similarity Measures on Multi-class Classification (x-axis: similarity measures, y-axis: fingerprinting methods)

risms of a base file (much like the power set file set). Base files are plagiarized and attack with saturation, naming, and redundancy attacks.

We can frame plagiarism detection in one of two ways: 1) binary classification or 2) multi-class classification. Binary classification is useful for practical use: it simply flags two files if a high enough similarity is detected. Multi-class classification offers more accurate estimates as to the type of plagiarism, and help protect against false positives.

To convert a similarity score to a category, we pick a noise threshold t that is an upper bound on the "coincidental" similarity between two source code files. This t can be varied for the application; we use a t of 0.30 based on the estimation that on average, common language features make up 30% of a source code file. For binary classification, it is as simple as if the score falls above or below the threshold. An examination of the correct threshold for different use cases has yet to be investigated.

For multi-class classification, each pair of files is given a ranking from 1 to 4:

1 - Not plagiarism

2 - Light plagiarism

3 - Heavy plagiarism

4 - Copy-paste plagiarism

Categories 2-4 map to the remaining $1-t$ in equal thirds.

In table 5, we compare our candidate fingerprint methods against similarity measures Jaccard (J), Sorensen-Dice (SD), Binary Cosine (BC), Sequenced Binary Cosine (SBC) for precision, recall, and f1-score.

4.2 Discussion

Observe that MOSS's k-gram method performs consistently across all similarity measures. Indeed the k-gram method consistently outperforms all other fingerprintng techniques. However, Merkle with no name outperforms MOSS on Sorensen-Dice Coefficient and Binary Cosine in precision. As previously discussed, Sorensen-Dice takes into account

	J	SD	BC	SBC
k-gram	0.77	0.77	0.77	0.69
Merkle	0.63	0.66	0.65	0.71
Merkle (no name)	0.69	0.74	0.74	0.37
Dyck	0.71	0.69	0.69	0.49
Dyck (no name)	0.71	0.69	0.69	0.34

Table 6: Recall of Fingerprint Methods vs Similarity Measures on Multi-class Classification (x-axis: similarity measures, y-axis: fingerprinting methods)

	J	SD	BC	SBC
k-gram	0.79	0.76	0.77	0.72
Merkle	0.60	0.59	0.59	0.71
Merkle (no name)	0.63	0.70	0.70	0.50
Dyck	0.70	0.69	0.67	0.56
Dyck (no name)	0.70	0.68	0.67	0.44

Table 7: F1-score for Fingerprint Methods vs Similarity Measures on Multi-class Classification (x-axis: similarity measures, y-axis: fingerprinting methods)

the size of fingerprint sets and is vulnerable to large saturation attacks (though it proves fairly robust on small saturation attacks in the corpus). Even so, Merkle no name still out performs MOSS on Binary Cosine which is robust to saturation attacks in precision.

Sequenced Binary Cosine, as previously discussed, is very sensitive to plagiarism. Unfortunately the similarities it gives are difficult to interpret: 0.70 can mean plagiarism, but so can 0.08. Even re-scaling the result is inconsistent at times. A better similarity score to measure sequence collisions remains a future prospect.

Recall across all similarity measures is lower; we suspect this is because the models are good at separating plagiarism from non-plagiarism, but the light and heavy categories are more challenging to classify. We expect better performance if framed as a binary classification problem. In general, as reported by the F1-score, Merkle (no name) performs second to MOSS.

5 Conclusion

We have proposed several alternatives to the state-of-the-art model in source code plagiarism detection: representing source code as an AST instead of raw text. We compared the performance of these alternatives with different similarity measures and found that many AST fingerprinting techniques are not robust against possible attacks. However, we found that with a Merkle-like fingerprinting method, we can improve precision across the most robust similarity measures. We also proposed our own novel extension to the Binary Cosine metric. While it did not outperform other metrics, we are confident it is a step in a promising direction.

There are many future opportunities. AST encoding was not exhaustively trialed; we suspect different tree traversals (e.g. in-order) might yield different results. Exploring alternatives to winnowing for fingerprint selection is also a interesting direction.

6 References

1. Alex Aiken, Daniel Wilkerson, Saul Schleimer. Winnowing: Local Algorithms for Document Fingerprinting. http://theory.stanford.edu/ aiken/publications/papers/sigmod03.pdf

2. Michel Chilowicz, Etienne Duris, Gilles Roussel. Syntax tree fingerprinting: a foundation for source code similarity detection. 2009. hal-00627811.

3. R.C. Merkle, "Protocols for public key cryptosystems," In Proc. 1980 Symposium on Security and Privacy, IEEE Computer Society, pages 122-133, April 1980.

4. Richard M. Karp and Michael O. Rabin. Pattern-matching algorithms. IBM Journal of Research and Development, 31(2):249260, 1987.

5. Jure Leskovec, Anand Rajaraman, Jeff Ullman. Mining of Massive Datasets. Cambridge University Press. Chapter 3. 2014.

6. Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In ICSE 07, pages 96105, Washington, DC, USA, 2007. IEEE-CS.