Integration of Static and Dynamic Code Stylometry Analysis for Programmer De-anonymization

Ningfei Wang
Lehigh University
niw217@lehigh.edu

Shouling Ji
[1]Zhejiang University
[2]Alibaba-ZJU Joint Research Institute of Frontier Technologies
sji@zju.edu.cn

Ting Wang
Lehigh University
inbox.ting@gmail.com

ABSTRACT

De-anonymizing the authors of anonymous code (i.e., code stylometry) entails significant privacy and security implications. Most existing code stylometry methods solely rely on static (e.g., lexical, layout, and syntactic) features extracted from source code, while neglecting its key difference from regular text – it is executable! In this paper, we present Sundae, a novel code de-anonymization framework that integrates both static and dynamic stylometry analysis. Compared with the existing solutions, Sundae departs in significant ways: (i) it requires much less number of static, hand-crafted features; (ii) it requires much less labeled data for training; and (iii) it can be readily extended to new programmers once their stylometry information becomes available.

Through extensive evaluation on benchmark datasets, we demonstrate that Sundae delivers strong empirical performance. For example, under the setting of 229 programmers and 9 problems, it outperforms the state-of-art method by a margin of 45.65% on Python code de-anonymization. The empirical results highlight the integration of static and dynamic analysis as a promising direction for code stylometry research.

CCS CONCEPTS

• **Security and privacy** → *Pseudonymity, anonymity and untraceability*;

KEYWORDS

Code Stylometry; Dynamic Analysis; De-anonymization

ACM Reference Format:
Ningfei Wang, Shouling Ji, and Ting Wang. 2018. Integration of Static and Dynamic Code Stylometry Analysis, for Programmer De-anonymization. In *11th ACM Workshop on Artificial Intelligence and Security (AISec '18), October 19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3270101.3270110

1 INTRODUCTION

It is known that, just like one's handwriting, every programmer has a distinct coding style, which is formed by a variety of preference choices: for example, some like to use `tab` over `space` for indentation, some prefer `for` loop over `while` loop, and some choose modular design over monolithic design.

This "code stylometry" allows one to fingerprint a programmer and to de-anonymize code that has been carefully sanitized. The practicality of code de-anonymization entails significant privacy and security implications especially in forensic contexts, including plagiarism detection, copyright dispute investigation, and malware authorship identification. For example, if an anonymous third-party software library matches a hacker's code stylometry, it is then sensible to assume that the library may not be safe to use.

The importance of code stylometry has attracted intensive research effort from the security communities. A plethora of code de-anonymization methods have been proposed [9, 17, 20, 26, 30, 35, 37, 41]. However, most existing methods use a large number of hand-crafted, *static* features (e.g., layout, lexical, and syntactic). For example, in [17], Caliskan-Islam *et al.* proposed to perform de-anonymization by performing dimensionality reduction on 120,000 static features and feeding the resulting features to a random forest classifier. Despite their impressive empirical performance, as they solely rely on static features, these methods are fairly sensitive to syntax changes and cannot be applied to compiled binary or assembly code.

Indeed, one prominent trait that distinguishes text and code is that code can be compiled and executed, which generates a large number of *dynamic* features, including memory dynamics, CPU dynamics, disk dynamics, and network dynamics. For example, some programmers may prefer to use `array` over `heap`, which can be captured by their memory access patterns. While such dynamic features are ignored in most existing work, we argue that they provide a distinct perspective on code stylometry. First, they well compliment static feature (details in Section 3). Second, they are insensitive to syntactic changes. Third, they are preserved in compiled binary and assembly code.

Our Work

Motivated by the drawbacks of existing work and the importance of dynamic features, in this paper, we present Sundae[1], the first-of-its-kind code de-anonymization framework that integrates both static and dynamic features in stylometry analysis.

The overall framework of Sundae is illustrated in Figure 1. Lying at the core of Sundae are three key components: (i) static feature extractor, which analyzes the given code in a static manner and extracts static features from it, (ii) dynamic feature extractor, which

[1]SUNDAE: Static UNderlined Dynamic Analysis Engine.

AISec '18, October 19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-6004-3/18/10…$15.00
https://doi.org/10.1145/3270101.3270110

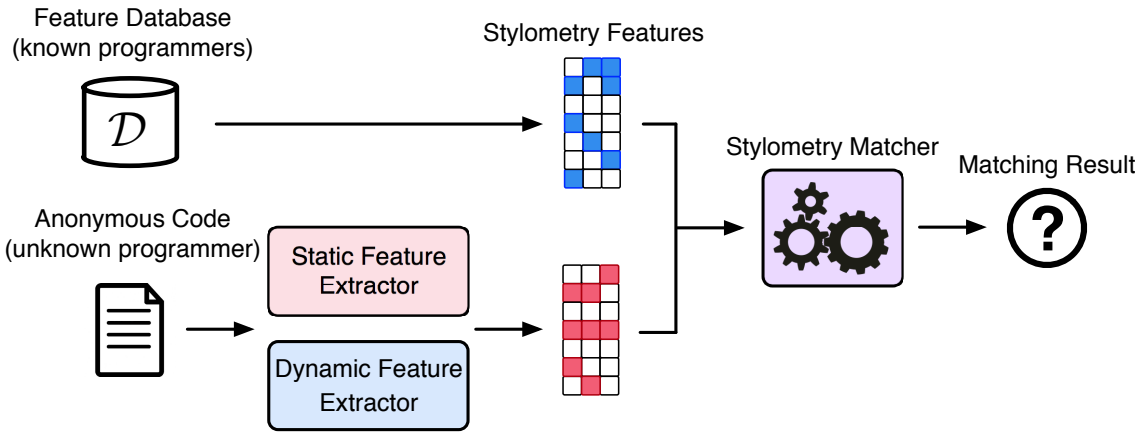

Figure 1: Overall framework of SUNDAE, which consists of three core components: static feature extractor, dynamic feature extractor, and stylometry matcher.

supplies necessary input, runs the code in an instrumented environment, and collects its dynamic features, and (iii) stylometry matcher, which compares the features of anonymous code with that of the known programmers in databases, and identifies the most probable matches.

To realize this framework, a number of challenges need to be carefully addressed, including: (i) the definition of dynamic features, (ii) the collection of dynamic features, and (iii) the extension to new programmers. To the first challenge, we define an array of dynamic feature based on three key aspects of the code execution: running time, memory access, and function call. To the second challenge, we employ a set of high-performance profilers to instrument source code and to monitor execution history. To the third challenge, we build a deep neural network (DNN)-based stylometry matcher, which follows the Siamese architecture and is able to readily incorporate new programmers once their stylometry information becomes available.

We implement SUNDAE for Python, which is one of the most popular programming languages and supported across different operating systems. However, it is worth emphasizing that while SUNDAE is implemented for Python, the underlying techniques can be readily generalized to other programming languages as well.

To empirically evaluate SUNDAE, we crawled the datasets of Python source code in Google Code Jam Competition from 2008 to 2017[1]. Across all the settings, SUNDAE outperforms the state-of-the-art code de-anonymization methods by large margins. For example, under the setting of 229 programmers and 9 problems, it outperforms the state-of-art method by a margin of 45.65% on Python code de-anonymization. The empirical results highlight the integration of static and dynamic analysis as a promising direction for code stylometry research.

Our Contributions

To the best of our knowledge, this work represents the first code de-anonymization framework that integrates both static and dynamic stylometry analysis. Our contributions can be summarized as follows:

- First, we articulate the drawbacks of existing work on code de-anonymization and highlight the importance of leveraging dynamic stylometry features in code de-anonymization.
- Second, we present SUNDAE, a novel framework which leverages both static and dynamic stylometry analysis and performs de-anonymization that is both accurate (i.e., it correctly identify the true authors of the anonymous code) and robust (i.e., it is insensitive to the number of candidate authors or the amount of labeled training data).
- Finally, through extensive evaluations on benchmark datasets, we show that SUNDAE outperforms the state-of-the-art methods in terms of both accuracy and robustness.

Our work is open-source and most of the source codes in this paper are available at https://github.com/ningfeiwang/Code_De-anonymization.

Roadmap

The remainder of the paper will proceed as follows. Section 2 introduces fundamental concepts and motivates the overall design of SUNDAE. Section 3 details the core components of SUNDAE, including static feature extractor, dynamic feature extractor, and stylometry matcher. Section 4 empirically evaluates the efficacy of SUNDAE in terms of de-anonymization accuracy and robustness. Section 5 discusses its limitations and points to several promising directions for further investigation. Section 6 surveys literature relevant to this work. The paper is concluded in Section 7.

2 OVERVIEW OF SUNDAE

In this section, we motivate the design of SUNDAE. At a high level, SUNDAE realizes two key ideas: (i) It integrates static and dynamic stylometry analysis, which significantly improves the accuracy and robustness of code de-anonymization. (ii) It adopts an extensible matching framework that readily incorporates the code stylometry of new programmers.

2.1 Integration of Static and Dynamic Analysis

Similar to written texts, it is found that computer programs also demonstrate a variety of style features unique to programmers [17], which enables static stylometry analysis of source code. The features used in literature can be roughly classified into three categories: *lexical* (e.g., use frequency of keywords such as `while`, `if` and `for`, the function number per file, and `print` styles), *layout* (e.g., the use of `tab` or four `space` for indentation), and *syntactic* (e.g., the characteristics of the abstract syntax tree of source code). Yet, relying solely on static analysis, existing methods often require a large number of hand-crafted, static features (e.g., 120,000 features in [17]).

Following this line of work, Sundae also extracts a collection of static features. However, thanks to its integration of dynamic analysis, Sundae only uses a small number (only 74 in our implementation) of general-purpose, static features found across a variety of programming languages.

While static stylometry analysis itself has achieved strong empirical performance [9, 17, 30, 35, 41], it suffers a set of fundamental limitations, including: (i) its sensitivity to syntactic change, (ii) its inapplicability to compiled or assembly code, and (iii) its requirement of a large number of hand-crafted features [33].

We argue that the drawbacks of static stylometry analysis can be greatly mitigated if it is integrated with *dynamic* stylometry analysis. Instead of focusing on its lexical, layout, or syntactic characteristics, dynamic stylometry analysis executes the source code, collects its runtime measurements (e.g., memory use, CPU use, data structure use), and identifies its dynamic, programmer-specific properties [11]. Note that dynamic stylometry analysis is invariant to syntactic change and applicable to compiled or assembly code, therefore complementary to static stylometry analysis.

However, to perform dynamic stylometry analysis, one needs to (i) instrument the source code with measurement mechanisms (e.g., breakpoints) and (ii) to supply valid input to effectively execute the code. We detail the implementation of measurement mechanisms and input generation in Section 3.

2.2 Extensible Code Stylometry Matching

We integrate the features generated by static and dynamic stylometry analysis using a deep neural network (DNN), motivated by their strong empirical performance in tasks that involve high-dimensional data. One straightforward way of using DNN in our setting is to construct a DNN-based classifier in which the number of classes is fixed as the number of known programmers in the dataset. This architecture, however, can not be easily extended: once the code stylometry of new programmers becomes available, it requires to re-train the whole system to incorporate the new data.

In Sundae, we adopt an extensible matching framework based on the Siamese architecture [16], which consists of two duplicate DNN networks, each extracting features from one input, and then computes their similarity (or distance). Because the Siamese network is decoupled from the features of specific programmers, once it is optimized on a small amount of training data, it can be readily applied to a much larger number of new programmers, without re-training the network. Further, adding new programmers to the database does not trigger the change of network architecture and can be implemented using computationally inexpensive, incremental training (details in Section 3).

3 DESIGN AND IMPLEMENTATION

In this section, we elaborate on the design and implementation of the three key components of Sundae, namely, static feature extractor, dynamic feature extractor, and stylometry matcher.

3.1 Static Feature Extractor

Follow the line of work on static stylometry analysis, we first define a set of static features, which can be extracted from source code without executing it. Specifically, we focus on three types of static features: lexical, layout, and syntactic. Yet, different from existing methods that use a large number of hand-crafted features, we only use a small number of general features that are found across different programming languages.

Table 1. List of Static Features (Lexical)

Feature	Description
Keywords	33 keywords (e.g., `while`, `if` and `for`)
Print	use of `()`
Comprehension	ratio of `for` within `[]`
Function number	log(# files/# functions)
Function length	average # LOC per function
Arguments	average # arguments per function
Import	ratio of `import`, `from import`, `import as`
Version	use of `Python 2` or `3` info. in comment
__name__	use of `__name__ == '__main__'`
Encoding	use of `UTF-8`

Lexical Features. Table 1 summarizes the set of lexical features used in Sundae. We detail a few important ones.

- *Keywords* – The frequencies of 33 keywords in code, including `for`, `while`, and `lambda`, demonstrate the programmer's preference for specific keywords. As an example, some prefer `for` over `while`.
- *Print* – As Python 3 requires parentheses in calling the `print` function (not mandatory in Python 2), this feature thus distinguishes the version of Python the programmer is more likely to use.
- *Comprehension* – Python provides comprehension as a concise way to construct lists and dictionaries. Its comparison with regular ways is shown in Figure 2. This feature measures the programmer's use of comprehension by checking the ratio of `for` that appears in `[]`.
- *Function number* – This feature measures the programmer's tendency of reusing code. Some prefer to implement and reuse similar operations in the form of functions, while others may reuse similar copy by copy-and-paste.
- *Function length* – The LOC per function is also a discriminative feature. To fulfill the same task, rather than a monolithic function, experienced programmers tend to use a number of modular functions, which are easier to debug.
- *Arguments* – The number of arguments per function indicates the "locality" of arguments: whether the programmer

```
a = [for x in range(10)]
```

```
a = []
for x in range(10):
    a += [x]
```

Figure 2: Two different ways of constructing lists in Python.

tends to define global variables, pass variables between functions, or confine computation within functions.

- *Import* – There are three ways of importing packages in Python: `import`, `from - import`, and `import - as`. This feature indicates the programmer's preference for each style.
- *Version* – The same things for 'Python 2 or 3' that check whether have some information of Python version in comments.
- *__name__* – Whether the code includes `if __name__ == '__main__'` indicates the programmer's tendency to make code importable and executable.
- *Encoding* – Whether the encoding (i.e., `UTF-8`) is included in comments is another feature that distinguishes mature and greenhorn programmers.

Table 2. List of Static Features (Layout)

Feature	Description
Indentation	Use of space or tab for indentation
Connection	use of space in string connection
Space	log(# words/# space)
Blank	log(# LOC/# blank lines)
Comment number	log(# LOC/# comments)
Comment length	average # words per comment
Line length	average # words per line

Layout Features. Table 2 summarizes the static, layout features used in SUNDAE. We detail each feature below.

- *Indentation* – We check whether the programmers use four spaces or one tab as indentation.
- *Connection* – We check whether the programmer uses a space on string connection (e.g., `a=1` versus `a = 1`).
- *Space* – This feature shows the percentage of space in code.
- *Blank* – The use of blank lines may also distinguish programmers: some use plenty of blank lines (e.g., each line of code followed by an empty line), while others are more stingy about the use of blank lines.
- *Comment number* – The comment code ratio indicates the programmer's preference of using comments in code. For example, some prefer to comment a large chunk of code, while others prefer to add short comments for each function.
- *Comment length* – This feature measure the average length of each comment.
- *Line length* – We also measure the average length of each line of code.

Table 3. Static Features (Syntactic)

Feature	Description
Height	AST tree height
Node	# nodes
Leave	# leaves
Var after 'for'	average length of variable after `for`
'_' after 'for'	average number of '_' after `for`
Initial	frequency of upper-case initials in variables
Upper	frequency of upper-case letters in variables
Lower	frequency of lower-case letters in variables
'_'	frequency of '_' in all variables

Syntactic Features. Following previous work, we also define the set of syntactic features based on the AST tree (abstract syntax tree) of the source code.

At a high level, AST is a tree-formed representation of the abstract syntactic structure of source code, wherein each node denotes a construct occurring in the source code, and each edge represents the relation between the corresponding two constructs. As an example, Figure 3 shows a piece of sample source code, and its corresponding AST tree. In our implementation, we use the AST module [3] to construct the AST tree and extract features from it.

We then extract a set of syntactic features from the AST tree, which are summarized in Table 3.

- *Height* – This feature reflects the deepest level the programmer places a node within the solution.
- *Node* – We measure the number of constructs (e.g., functions, variables, operators) in the code.
- *Leave* – We measure the number of leave nodes in the tree.
- *Var, '_' after 'for'* – These features measure the average length of variables and the average number of '_' in variables after `for`.
- *Initial, upper, lower, and '_'* – These features capture the patterns in the variable names, such as the frequencies of upper-case, lower-case, and '_' in variables.

All the features in Table 1, 2, and 3 are extracted from the source code and encoded into a feature vector, which forms the static features of the code of interest.

3.2 Dynamic Feature Extractor

One significant distinction of SUNDAE from the state-of-the-art code de-anonymization methods is its incorporation of dynamic stylometry analysis. In SUNDAE, we collect the characteristics observed during executing the source code and construct a set of dynamic features to distinguish programmers.

Specifically, we focus on three main categories of dynamic features, running time, memory, and disassembled code. For the running time, we use the Python profiler cProfile [7] to measure the running time of individual modules. For memory usage, we use the Python profiler memory_profiler [6] to measure the memory usage of individual functions. For disassembled code, we use the Python module dis to disassemble the Python bytecode back to the Python source codes [5].

The list of features is summarized in Table 4. The details are given below.

```
1 def fun(a,b):
2         return a * b
3
4 fun(2,3)
```

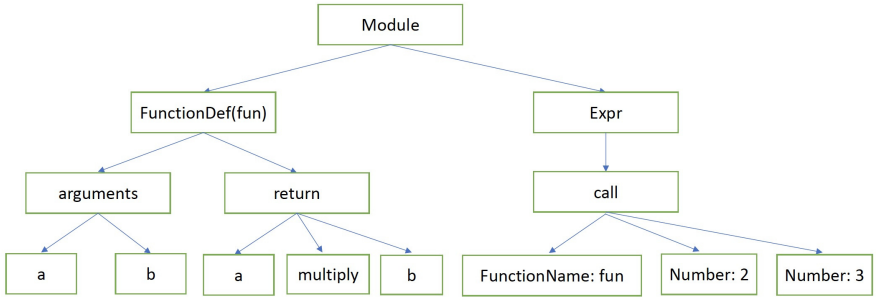

Figure 3: Sample python source code and its AST tree.

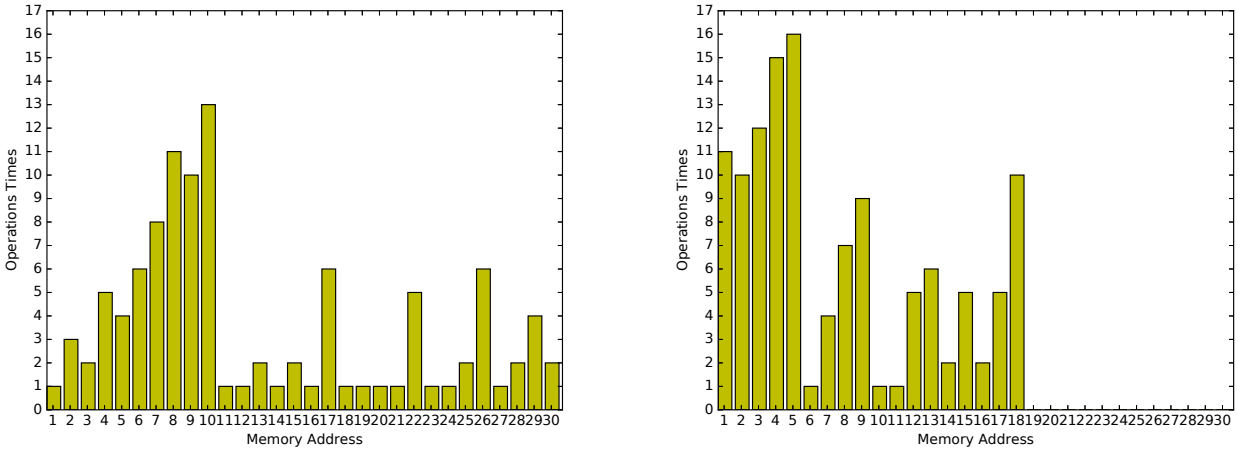

Figure 4: Memory access patterns of the code of two distinct programmers for the same Google CodeJam problem ("Oversized Pancake Flipper" - problem id: "5736519012712448", competition id: "3264486").

Table 4. Dynamic Features

Feature	Description
Function call	# function calls
Module time	average running time per module
Total time	total running time of
Function memory	memory usage of each function
Total memory	total memory usage
Memory access	memory access patterns
Disassembled code	length of disassembled code

- *Function call* – This feature measures the usage frequencies of customized and built-in functions (e.g., `range`, `append`, `iter`). This feature can determine whether the programmers prefer using functions (including functions defined by programmers and Python internal functions) to achieve specific objectives. For example, to append values into a list in Python, the programmer may use `append` function or use list addition operations.
- *Module time* – This features measures the running time of individual modules, which suggests whether the programmer prefers to use a number of modules, each with short running time, or integrate different modules inside a large one with long running time.
- *Total time* – This feature measures the total running of the code, which reflects the inherent complexity of the code for the given problem.
- *Function memory* – We collect the average memory usage of each function when it is invoked, which reflects the programmer's fine-grained memory usage patterns.
- *Total memory* – We also measure the total memory usage per file. This feature address the memory usage habits roughly (e.g., `free the memory after managing`).
- *Memory access* – This feature details the memory access patterns of the given code, in the form of histogram (frequency versus address). To show the effectiveness of this feature, we randomly pick a problem in the Google CodeJam competition as an example ("Oversized Pancake Flipper" [2]). Figure 4 contrasts the memory access patterns of the code by two programmers. It is clear that the first programmer uses

a large spectrum of memory addresses, while the second one intensively utilizes a small range of memory addresses. We thus use memory access patterns as a discriminative feature to distinguish programmers. To be computationally efficient, we create the feature values by extracting the histogram values for specific frequencies (e.g., 10 times).

- *Disassembled code* – The original Python code may be redundant and deviate far from its assembled version. To count this difference, we disassemble the bytecode of the original code back to source code and measures its length.

All the features in Table 4 are extracted from the source code and encoded into a feature vector, which in conjunction of the set of static features, form the complete feature set.

Another challenge for extracting the dynamic features is to supply valid input to effectively execute the source code. The discussion on generating valid inputs is deferred to Section 4.

3.3 Stylometry Matcher

In SUNDAE, we build the stylometry matcher based on a deep neural network (DNN) architecture. However, traditional DNNs used in classification tasks assume a fixed number of classes (i.e., a fixed number of programmers in our setting), which significantly hinders its extension to new programmers in our setting.

Instead, we build the stylometry matcher using a Siamese architecture, which takes as input two (raw) feature vectors (e.g., the features of anonymous code, and the features of a known programmer) and outputs the similarity score of these two feature vectors, indicating whether the anonymous code is likely authored by this known programmer. It is noticed that as this architecture is not coupled with the number of classes in the dataset, it can be readily extended to new programmers as their stylometry becomes available.

As shown in Figure 5, in a nutshell, the Siamese network consists of three major parts, two base networks, and one matcher. The two identical base networks are responsible for abstracting high-level feature vectors from the input feature vectors, and the matcher is responsible for computing the similarity score of the two high-level feature vectors.

In current implementation, we adapt the basic Siamese architecture to the task of code de-anonymization. The base network is a pre-trained DNN model with four fully-connected layers, with detailed architecture given in Table 5. The matcher is formed by a subtract layer, a fully connected layer, and a softmax layer. The outputs of the two base networks are fed to the matcher, which outputs the similarity score of the two inputs (i.e., how likely the anonymous code is authored by the known programmer).

Table 5. Architecture of the base networks used in SUNDAE.

Property	Parameters
Hidden Layers Number	4
Dropout Rate	0.2
Hidden Layers Neural Numbers	400, 300, 200, 200
Active function	"Relu"
Batch Size	50
Epoch	20

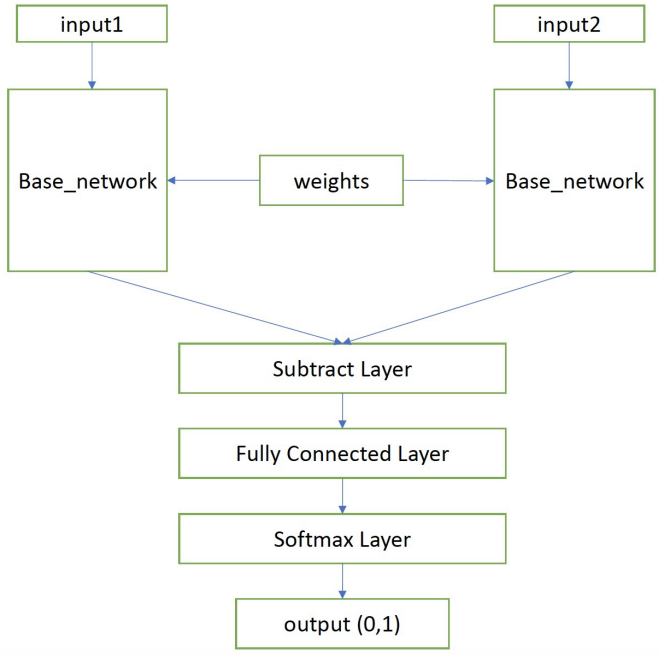

Figure 5: Illustration of a Siamese Network, which consists of three core parts, two identical base networks and one matcher.

4 EMPIRICAL EVALUATION

In this section, we empirically evaluate the proposed SUNDAE framework. The experiments are designed to answer two key questions: (i) How effective is SUNDAE in terms of de-anonymizing code, compared with alternative methods? (ii) What is the distinguishing power of different features selected by SUNDAE?

We begin with describing the experimental setting.

4.1 Experimental Setting

Datasets. Following previous work [9, 17], we use the source code in the Google CodeJam competition archive [1] as the benchmark dataset in our evaluation. This competition calls on programmers around the world to solve a set of algorithmic problems, with participants of different educational levels (e.g., middle-school students, high-school teachers, college professors). Thus, the dataset covers a large demographical diversity.

We build a customized crawler to simulate a Chrome Browser. We let the crawler visit the Google CodeJam website and downloads the dataset of archived competitions (2008 to 2017), which includes the following key parts: the competition questions, the sample inputs to the questions, the source code (e.g., solutions) submitted by participants, and the ids of the participants.

We then divide the resulting dataset into different subsets based on the LOC of source code of each programmer. Specifically, the dataset is partitioned into four subsets, as summarized in Table 6. For example, the first dataset (D80) includes the programmers (totally 75 programmers) who have contributed more than 80K LOC to the competition, which involve 221 problems. It is noted that from

Table 6. Statistics of the Google CodeJam dataset.

Dataset	LOC/Programmer	#Programmers	#Problems
D80	>80k	75	221
D60	>60k	139	223
D40	>40k	330	224
D20	>20k	1,159	229

dataset1 to dataset4, due to the growing number of programmers and the decreasing amount of code per programmer, it becomes increasingly challenging to de-anonymizing the code.

For the D80, D60, and D40 datasets, we partition each dataset as 80% for training (20% of the training data for validation) and 20% for testing. The D20 dataset is the most challenging setting, in which we train the code de-anonymization models on the D60 dataset and apply the trained model to the entire D20 dataset for testing.

Table 7. Running environment to collect dynamic features.

Property	Configuration
CPU	Intel-Core i7-6700K @4.00GHz x 8
memory	16GB
OS	Ubuntu 18.04 LTS
Anaconda	5.1.0

Running Environment. As SUNDAE requires to analyze the dynamic features of the given code during its execution, the results thus highly depend on the hardware platform, especially for running time and memory usage-related measurements. We thus dedicate a standalone workstation to the collection of dynamic features. Also, as the execution of Python code often requires a number of external libraries, we use Anaconda [4] to supply these libraries. Anaconda include 250+ popular data science packages for Python, which is sufficient for our evaluation.

The details of the hardware and software environments are summarized in Table 7.

4.2 Comparative Evaluation

End-to-End Performance. In the first set of experiments, we compare SUNDAE with a set of state-of-the-art code de-anonymization methods in terms of their end-to-end performance of code de-anonymization.

Specifically, Aylin et al. [17] and Bander et al. [9] also perform code de-anonymization on the Python code in the Google CodeJam dataset. We thus directly quote their reported results. In addition, we re-implement [21], which employs C4.5 decision tree as the classifier and includes all the features used in [21] and available in Python. Finally, we re-implement [25], another state-of-the-art code de-anonymization method, which uses a nearest neighbor classifier and includes all the features used in [21] and available in Python. We replicate the setting reported in the original papers using our dataset.

The results are listed in Table 8 and Figure 6. We have the following key observations. First, across all the settings, SUNDAE outperforms existing methods in terms of de-anonymization accuracy, even though it uses much less number of static features compared

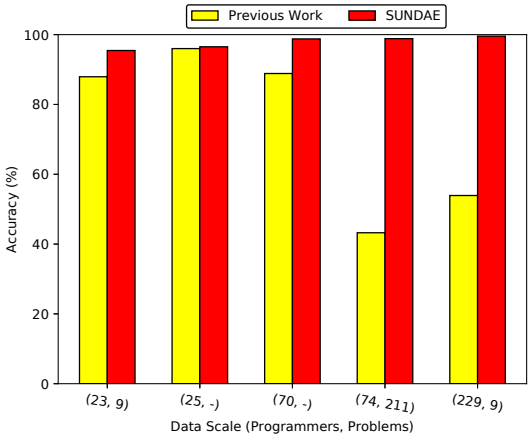

Figure 6: Comparison of SUNDAE and alternative methods under different scales.

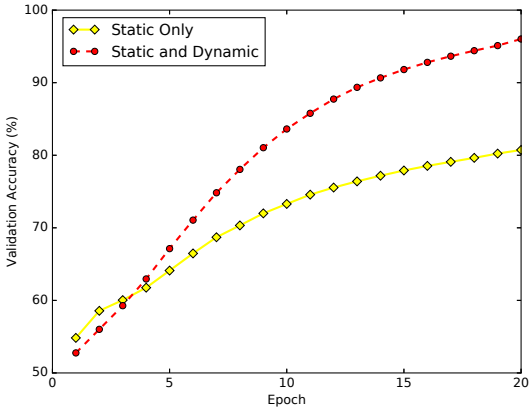

Figure 7: Comparison of the performance of SUNDAE and its variant based on static features.

with alternative methods. For example, in the case of 229 programmers and 9 problems, SUNDAE leads [17] by a margin of 45.65%! Second, across all the settings, SUNDAE attains fairly stable accuracy, varying from 95.45% to 99.56%. Both phenomena can be attributed to the unique framework of SUNDAE, which integrates both static and dynamic stylometry analysis, and overcomes the instability limitations of static features.

In addition, we further evaluate SUNDAE on all the datasets in Table 6, with results shown in Table 9. It is interesting to notice that SUNDAE is insensitive to the amount of code available for each programmer and the number of programmers. Indeed, the accuracy slightly increases from 98.84% to 99.91% from D80 to D20. This can be attributed to the extensibility nature of the stylometry matcher, which can be readily extended to new programmers, without affecting its performance on existing ones.

Table 8. Comparison of SUNDAE and alternative code de-anonymization methods.

Method	#Programmers	#Problems	Accuracy	Accuracy (SUNDAE)
Elenbogen *et al.* [21]	74	221	20.81%	**98.84%**
Frantzeskou *et al.* [25]	74	221	43.24%	**98.84%**
Caliskan-Islam *et al.* [17]	23	9	87.93%	**95.45%**
Caliskan-Islam *et al.* [17]	229	9	53.91%	**99.56%**
Alsulami *et al.* [9]	25	-	96.00%	**96.52%**
Alsulami *et al.* [9]	70	-	88.86%	**98.77%**

Table 9. Performance of SUNDAE on the Google CodeJam datasets.

Method	Dataset	#Programmers	#Problems	#Accuracy
SUNDAE	D80	74	221	98.84%
	D60	139	223	99.24%
	D40	331	224	99.68%
	D20	1,159	229	99.91%

Static versus Dynamic Analysis. To further understand the impact of dynamic stylometry analysis, in the second set of experiments, we create a variant of SUNDAE, that uses only the static features.

Figure 7 shows the de-anonymization accuracy of SUNDAE and its variant as a function of the number of training epochs. It is observed that the use of dynamic stylometry analysis not only significantly boosts the overall accuracy but also improves the training efficiency. For example, at the end of the 10-th epoch, SUNDAE already attains 84% accuracy, while its variant based on static features only reaches 72% accuracy. We can thus conclude that incorporating dynamic features seems a promising direction for further improving code stylometry analysis.

Figure 8: Distribution of the importance scores of features selected by SUNDAE.

4.3 Feature Evaluation

In this set of experiments, we perform an ablation study on the importance of the features selected by SUNDAE.

Most existing work uses information theoretical metrics (e.g., mutual information) to measure feature importance (e.g., by computing the mutual information of features and the ground-truth classes). Yet, in the setting of SUNDAE, as we adopt a deep neural network (DNN)-based architecture, we use a DNN-based metric to meaure feature importance.

Specifically, we examine the base network in SUNDAE (recall that both base networks are identical) and measure the importance of a feature by the (absolute) average weights on the 400 outgoing connections of the corresponding input neuron in the DNN. The distribution of the importance scores of features selected by SUNDAE are shown in Figure 8. It is observed that the distribution follows a long-tail distribution, indicating that there are a few important features and a large number of less important ones.

Table 10 and Table 11 list respectively the 10 most important features and the 5 least important ones. Observe that the top 10 list includes both static and dynamic features (while the bottom 5 list consists of only static features), indicating that the dynamic features well complement the static ones in SUNDAE.

Table 10. The 10 most important features selected by SUNDAE.

Rank	Feature	Weights (ABS)
1	' '	0.6397
2	Print	0.5790
3	Indentation	0.5559
4	Upper	0.4258
5	Function number	0.4044
6	Node	0.3943
7	Memory access	0.3919
8	Function length	0.3813
9	Arguments	0.3598
10	Function Memory	0.3359

4.4 Extensibility Evaluation

In the final set of experiments, we evaluate the extensibility of SUNDAE, i.e., how SUNDAE can incorporate new programmers once their stylometry information becomes available. Specifically, we compare the training time of SUNDAE and alternative architectures when adding a new programmer into the database. To make a fair comparison, we construct the alternative architecture as the

Table 11. The 5 least important features selected by SUNDAE.

Rank	Feature	Weights (ABS)
1	Keywords ("return")	0.0039
2	Keywords ("in")	0.0040
3	Keywords ("is")	0.0049
4	Keywords ("def")	0.0085
5	Keywords ("if")	0.0094

integration of the "Base_network" and a "softmax" layer. The update time is defined as the total running time until convergence after adding a new programmer into the database.

Figure 9 shows the detailed comparison. Observe that the update time of SUNDAE is fairly insensitive to the number of programmers in the database. In comparison, the update time of the alternative architecture increases approximately linearly with the number of programmers. This difference may be explained as follows. SUNDAE essentially learns the metric to measure the distance between different programmers, while the addition of new programmers may have fairly limited impact to this metric. In contrast, the alternative architecture is essentially a multi-class classifier, in which the number of classes equals to the number of programmers in the database; thus, adding a new programmer needs to adjust the distribution among all the programmers in the database.

Thereby, we can conclude that SUNDAE can be readily extended to new programmers.

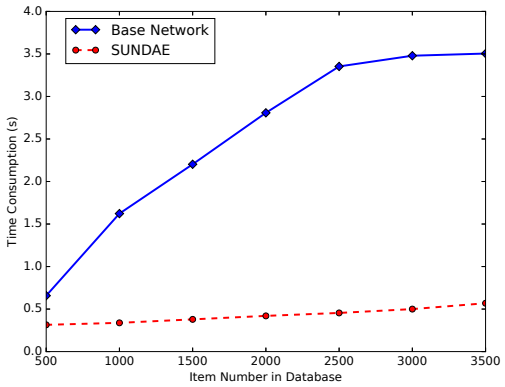

Figure 9: Extensibility comparison between SUNDAE and alternative architecture.

5 DISCUSSION

In this section, we discuss the limitations of this work, possible countermeasures, and several directions for further research.

5.1 Limitations

First, while the Google CodeJam dataset used in our evaluation show realistic features (e.g., strong sparsity – only 75 programmers have contributed 80K+ LOC), it is still significantly different from real coding environments. For instance, the CodeJam dataset focuses on 229 well-defined problems with sample inputs and standard outputs, while in real settings, often the tasks are much more complicated (i.e., consisting of multiple problems).

Second, the de-anonymization accuracy varies significantly with the nature of the problems. The Google CodeJam competition provides multiple difficulty levels, which manifests in different rounds. We observe that the difficulty level often impacts the distinctive patterns of different programmers. For example, if the problem is to implement `Quicksort` with an array, the programmers tend to consider two ways, recursion and non-recursion, and some programmers may use the same template codes; if the problem is to implement `Quicksort` with a linked list, the programmers show a large variation in their coding styles, from double linked list, to single linked list, and that will provide more distinct features to fingerprint the programmer.

Third, we do not consider the cases that the source code is designed by multiple programmers. Large projects on platforms such as `Github` and `Gitee` often involve plenty of contributors. In order to handle such cases, code fragment analysis is indispensable to localize the fragments contributed by different programmers. Nevertheless, to execute code fragments and to extract dynamic features, more instrumentation and input generation are needed. For example, if we split the source code into individual functions, we need to detect the input data types (e.g., `list` and `dictionary`) and feed precise inputs to ensure execution.

Fourth, we do not consider the cases of malicious source code. Such code, if executed, may harm the operating systems or even hardware. For such cases, we often need sandbox environments (e.g., virtual machines) to run the source code to extract dynamics features.

5.2 Countermeasures

Next, we discuss possible countermeasures that may defend against our de-anonymization attacks.

First, there are integrated development environments (IDEs) that include normalization tools (e.g., `vim` includes adjusting indentation methods), which may help unify coding styles. With the help of such tools, many static features in Section 3 may be less discriminative, which may influence the de-anonymization accuracy.

Second, it is possible to inject noise into source code to mislead the de-anonymization method. For example, one can pad a random number of `enter` to comments, which may influence the feature of average comment length.

5.3 Future Directions

This work also opens several research directions that are worth further investigation.

First, we consider extending our approaches – integrating static and dynamic features – to other programming languages (e.g., C, C++ and Java) as our ongoing research. Note that some static features (lexical) which are specific to Python need to be re-defined for other programming languages is required, while most dynamic features are universal across different programming languages.

Second, the feature extraction in SUNDAE is mostly hand-crafted. We consider applying machine learning (especially deep learning) techniques to automate this process. For example, in previous work, Alsulami *et al.* [9] manage to apply Long Short-Term Memory(LSTM) to train the AST trees, which represents a novel use of DNN in stylometric feature engineering.

Third, we believe that SUNDAE may help bug tracing and remedy recommendation. After obtaining the authorship of the code, one may build a database to create profiles for programmers about their tendency to inject bugs. Then, once a potential bug is reported, one may easily find the likely responsible programmers.

Fourth, large projects, especially open source projects, may involve more than one contributor. It is an interesting direction to extend SUNDAE to handle the cases of multiple authors by integrating it with techniques such as code fragmentation.

6 RELATED WORK

In this section, we review the relevant literature in four categories: stylometry, source code de-anonymization, static and dynamic analysis, and classification model.

6.1 Stylometry

Traditionally stylometry is mainly applied in linguistic fields. Bergsma *et al.* [13] present an approach to automatically recover hidden attributes of scientific articles (e.g., whether the author is male or female). Feng *et al.* [22] investigate syntactic stylometry for deception detection. Brocardo *et al.* [15] perform analysis for authorship verification in short texts using stylometry. To defend against such de-anonymization attacks, there is also work on possible anonymization countermeasures [8, 14, 32].

6.2 Code De-anonymization

One of the most popular de-anonymization methods is based on the syntactic features, which has shown significant success in source code authorship attribution [9, 17, 30, 35, 41]. Typically, an AST tree [34] is built based on the source code and the code authorship is detected by analyzing the AST tree. For example, Alsulami *et al.* [9] use Long Short-Term Memory(LSTM) based on the AST trees of Python and C++ source code, and achieve 88.86% in the case of 70 programmers on Python. In addition, N-grams (e.g., frequencies of tokens in source code) [24, 25] and layout and lexical features [17, 20, 26] are also popular features used in syntactic feature-based code de-anonymization. In this line of work, Caliskan-Islam *et al.* [17] achieve 98.04% accuracy under the setting of 250 programmers and 9 problems for C++, and 87.93% accuracy under the setting of 23 programmers and 53.91% under the setting of 229 programmers for Python. Ding *et al.* [20] achieve 67.2% accuracy under the setting of 46 programmers for Java.

Another line of work extends code de-anonymization to binary codes. Rosenblum *et al.* [36] propose methods of detecting the stylistic features from binary codes. Caliskan-Islam *et al.* [18] use disassembly and decompilation techniques to process binary source code and extract features in disassembly and decompilation code. They input all the features into a random forest classifier and obtain highly accurate de-anonymization for C++.

In addition, Kothari *et al.* [29] propose a set of distance metrics on unidentified source code to determine the closest matching profiles. Specifically, they create the profile database and compute the distance of the testing data to the data from the database. Spafford *et al.* [37] proposed to use lexical features and syntactic features in source code authorship detection.

6.3 Static and Dynamic Analysis

Static analysis [33] analyzes the behaviors of computer programs without actually executing them. Nowadays, static analysis has been widely applied in software engineering. For example, it is used to predict the worst-case execution time [23] or to detect the error of source code without running it [39].

Dynamic analysis [11] analyzes the properties of running programs. It has been used in correcting the bugs of programs automatically [40]. Hoorn *et al.* [38] introduce an extensible framework for monitoring and analyzing the runtime behavior of software systems.

Further, there are prior works that attempt to combine static and dynamic analysis. Bacci *et al.* [10] measure the robustness to code obfuscation via static and dynamic analysis. Hu *et al.* [27] proposed DUET, a framework which integrates dynamic and static analyses for malware clustering. Chen *et al.* [19] propose an approach that integrates static and dynamic analysis to find subtle atomicity violations.

6.4 Classification Models

Several classification models have been widely used in code stylometry analysis. Random forest [31] has been used in [17, 18, 41] and achieves impressive accuracy. C4.5 decision tree is another model used in code de-anonymization. Elenbogen *et al.* [21] use C4.5 decision tree to detect outsourced students programming assignments in 12 programmers and achieve 74.4% accuracy. Nearest neighbor classifier has been used in de-anonymizing java code [25].

Nowadays, deep learning models have achieved tremendous success in a range of applications such as image classification and natural language processing. The Siamese network is a special type of neural network architecture. Instead of learning to classify the inputs, the Siamese network learns to differentiate two inputs, i.e., the similarity or distance. It has been applied in a number of applications. For example, Bromley *et al.* [16] use the Siamese network in signature verification; Baraldi *et al.* [12] use the Siamese network in scene detection; and Koch *et al.* [28] perform image classification using Siamese network.

7 CONCLUSION

In this paper, we present the design, implementation, and evaluation of SUNDAE, a novel code de-anonymization framework that integrates both static and dynamic stylometry analysis. Through extensive evaluation on benchmark datasets, we demonstrate that compared with existing code de-anonymization methods, SUNDAE enjoys a set of key advantages: (i) it only requires a small number of hand-crafted features; (ii) it is insensitive to the number of programmers in the pool; and (iii) it is extensible to new programmers without re-training. The empirical results indicate the integration of static and dynamic analysis as a promising avenue for further code stylometry research.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1566526 and 1718787. Shouling Ji is partly supported by NSFC under No. 61772466, the Provincial Key Research and Development Program of Zhejiang, China under No.

2017C01055, the Alibaba-ZJU Joint Research Institute of Frontier Technologies, the CCF-NSFOCUS Research Fund under No. CCF-NSFOCUS2017011, and the CCF-Venustech Research Fund under No. CCF-VenustechRP2017009.

REFERENCES

[1] 2017. Google Code Jam. https://code.google.com/codejam/ Google Code Jam link.

[2] 2017. Google Code Jam contest 3264486. https://code.google.com/codejam/contest/3264486/dashboard Google Code Jam link.

[3] 2018. Abstract Syntax Trees. https://docs.python.org/2/library/ast.html Python AST module.

[4] 2018. Anaconda. https://www.anaconda.com/ Anaconda.

[5] 2018. Disassembler for Python bytecode. https://docs.python.org/3/library/dis.html Python dis module.

[6] 2018. memory profile. https://pypi.org/project/memory_profiler/ Python memory profile.

[7] 2018. Python cProfile. https://docs.python.org/2/library/profile.html Python cProfile.

[8] Sadia Afroz, Michael Brennan, and Rachel Greenstadt. 2012. Detecting hoaxes, frauds, and deception in writing style online. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 461–475.

[9] Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. 2017. Source Code Authorship Attribution Using Long Short-Term Memory Based Networks. In *European Symposium on Research in Computer Security*. Springer, 65–82.

[10] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. 2018. Impact of code obfuscation on android malware detection based on static and dynamic analysis. In *4th International Conference on Information Systems Security and Privacy*. Scitepress, 379–385.

[11] Thomas Ball. 1999. The concept of dynamic analysis. In *Software Engineering?ESEC/FSE?99*. Springer, 216–234.

[12] Lorenzo Baraldi, Costantino Grana, and Rita Cucchiara. 2015. A deep siamese network for scene detection in broadcast videos. In *Proceedings of the 23rd ACM international conference on Multimedia*. ACM, 1199–1202.

[13] Shane Bergsma, Matt Post, and David Yarowsky. 2012. Stylometric analysis of scientific articles. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 327–337.

[14] Michael Brennan, Sadia Afroz, and Rachel Greenstadt. 2012. Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity. *ACM Transactions on Information and System Security (TISSEC)* 15, 3 (2012), 12.

[15] Marcelo Luiz Brocardo, Issa Traore, Sherif Saad, and Isaac Woungang. 2013. Authorship verification for short messages using stylometry. In *Computer, Information and Telecommunication Systems (CITS), 2013 International Conference on*. IEEE, 1–6.

[16] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1994. Signature verification using a" siamese" time delay neural network. In *Advances in Neural Information Processing Systems*. 737–744.

[17] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium (USENIX Security), Washington, DC*.

[18] Aylin Caliskan-Islam, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. 2015. When coding style survives compilation: De-anonymizing programmers from executable binaries. *arXiv preprint arXiv:1512.08546* (2015).

[19] Qichang Chen, Liqiang Wang, Zijiang Yang, and Scott D Stoller. 2009. HAVE: detecting atomicity violations via integrated dynamic and static analysis. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 425–439.

[20] Haibiao Ding and Mansur H Samadzadeh. 2004. Extraction of Java program fingerprints for software authorship identification. *Journal of Systems and Software* 72, 1 (2004), 49–57.

[21] Bruce S Elenbogen and Naeem Seliya. 2008. Detecting outsourced student programming assignments. *Journal of Computing Sciences in Colleges* 23, 3 (2008), 50–57.

[22] Song Feng, Ritwik Banerjee, and Yejin Choi. 2012. Syntactic stylometry for deception detection. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers-Volume 2*. Association for Computational Linguistics, 171–175.

[23] Christian Ferdinand and Reinhold Heckmann. 2004. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*. Springer, 377–383.

[24] Georgia Frantzeskou, Stephen MacDonell, Efstathios Stamatatos, and Stefanos Gritzalis. 2008. Examining the significance of high-level programming features in source code author classification. *Journal of Systems and Software* 81, 3 (2008), 447–460.

[25] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. 2006. Source code author identification based on n-gram author profiles. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*. Springer, 508–515.

[26] Andrew Gray, Stephen MacDonell, and Philip Sallis. 1997. Software forensics: Extending authorship analysis techniques to computer programs. (1997).

[27] Xin Hu and Kang G Shin. 2013. DUET: integration of dynamic and static analyses for malware clustering with cluster ensembles. In *Proceedings of the 29th annual computer security applications conference*. ACM, 79–88.

[28] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. 2015. Siamese neural networks for one-shot image recognition. In *ICML Deep Learning Workshop*, Vol. 2.

[29] Jay Kothari, Maxim Shevertalov, Edward Stehle, and Spiros Mancoridis. 2007. A probabilistic approach to source code authorship identification. In *Information Technology, 2007. ITNG'07. Fourth International Conference on*. IEEE, 243–248.

[30] Flavius-Mihai Lazar and Ovidiu Banias. 2014. Clone detection algorithm based on the Abstract Syntax Tree approach. In *Applied Computational Intelligence and Informatics (SACI), 2014 IEEE 9th International Symposium on*. IEEE, 73–78.

[31] Andy Liaw, Matthew Wiener, et al. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.

[32] Andrew WE McDonald, Sadia Afroz, Aylin Caliskan, Ariel Stolerman, and Rachel Greenstadt. 2012. Use fewer instances of the letter âĂIJiâĂİ: Toward writing style anonymization. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 299–318.

[33] Anders Møller and Michael I Schwartzbach. 2012. Static program analysis.

[34] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. 2005. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes* 30, 4 (2005), 1–5.

[35] Brian N Pellin. 2000. Using classification techniques to determine source code authorship. *White Paper: Department of Computer Science, University of Wisconsin* (2000).

[36] Nathan Rosenblum, Xiaojin Zhu, and Barton P Miller. 2011. Who wrote this code? identifying the authors of program binaries. In *European Symposium on Research in Computer Security*. Springer, 172–189.

[37] Eugene H Spafford and Stephen A Weeber. 1993. Software forensics: Can we track code to its authors? *Computers & Security* 12, 6 (1993), 585–595.

[38] André Van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM, 247–248.

[39] Kostyantyn Vorobyov and Padmanabhan Krishnan. 2010. Comparing model checking and static program analysis: A case study in error detection approaches. *Proceedings of SSV* (2010).

[40] Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic Neural Program Embedding for Program Repair. *CoRR* abs/1711.07163 (2017). arXiv:1711.07163 http://arxiv.org/abs/1711.07163

[41] Wilco Wisse and Cor Veenman. 2015. Scripting dna: Identifying the javascript programmer. *Digital Investigation* 15 (2015), 61–71.