Does Coding Style Really Survive Compilation? Stylometry of Executable Code Revisited

Muaz Ali
The University of Arizona
Tucson, AZ, USA
muaz@arizona.edu

Tugay Bilgis*
The University of Arizona
Tucson, AZ, USA
tbilgis@arizona.edu

Nimet Beyza Bozdag*
The University of Arizona
Tucson, AZ, USA
nbbozdag@arizona.edu

Saumya Debray
The University of Arizona
Tucson, AZ, USA
debray@arizona.edu

Sazzadur Rahaman
The University of Arizona
Tucson, AZ, USA
sazz@arizona.edu

ABSTRACT

This paper describes a replication study of influential recent work on binary-level code stylometry by Caliskan *et al.* [8]. Using the Google Code Jam (GCJ) dataset that the original work used but with possible differences in authors and tasks, the accuracy results we obtain are significantly lower than those originally reported. An analysis of the features that contribute most to author classification decisions indicates that many such features are accidental artifacts arising from the erroneous disassembly of data bytes embedded in the binary and have little to do with programming style. Our results suggest that binary-level code stylometry. (1) is more sensitive to code characteristics than previously suspected; (2) can be significantly less accurate than previously reported (for 100 authors, we achieved approximately 63% accuracy, compared to the 96% reported in the original work); and (3) deserves careful attention to accidental artifacts arising from the compilation and stylometry toolchains. We found 29/33 of the top `ndisasm`-based features are the result of erroneous disassembly. Our analysis shows that this can cause the model to pick spurious features and thereby unknowingly inflate the results.

1 INTRODUCTION

Stylometry refers to the analysis of the style characteristics of individual authors to identify the authorship of documents. It has been used for authorship attribution of a diverse range of works, including natural language texts [21, 32], music [9, 46], and art [20, 33]. There has also been a considerable body of work on stylometry applied to software, focusing on both source code [5, 22, 40, 44, 45] as well as executable binaries [4, 8, 38].

Code stylometry has many legitimate uses. At the source code level, it can be useful for detecting plagiarism and resolving copyright disputes; at the binary level, it can help identify the authors of malicious software. However, it can also pose a profound threat to the privacy of programmers who legitimately wish to remain anonymous. For example, programmers who develop software to get around censorship or surveillance tools deployed by repressive regimes [37] or organize protests or engage in online activism [15, 23, 43] have reason to fear for their safety and well-being should their authorship of such software be unmasked. Conversely, errors in code authorship attribution may falsely implicate people who have nothing to do with the software in question. It is therefore important to understand the capabilities and limitations of code stylometry and circumstances that can impact its accuracy and robustness. Since software is often distributed in binary form, it is especially important to address these issues in the context of binary-level code stylometry. Importantly, we are not concerned with code obfuscation or adversarial techniques aimed explicitly at hindering stylometry [19, 34]: our goal is to investigate stylometry as applied to code written by "ordinary" programmers using "ordinary" software development processes.

This paper explores the replicability and robustness of binary-level code stylometry techniques proposed in an influential recent paper by Caliskan *et al.* [8]. We used the code and experimental setup publicly shared by the authors. We did not have access to the datasets originally used for training and testing since these were not shared due to privacy considerations, but we closely followed the paper's description of how they were constructed. In particular, we followed Caliskan *et al.* [8] in using Google Code Jam submissions. We used the same criteria as in their paper to select authors and their submitted code samples. However, the authors and tasks we sampled from the data might differ from that of the original work because we do not have access to the original dataset. To ensure fidelity when compiling these programs to executables and extracting features from the executables, we used the same tools as in their paper; where possible, we used information from the GitHub repository specified in the paper to use the same versions of the tools and invoked them with the same command-line arguments. Overall, this replication study answers the following research questions.

- **RQ1 (Reproducibility study)–§5:** To what extent are binary code authorship attribution results from [8] reproducible and replicable to other settings?
- **RQ2 (Cause Analysis)–§6:** What are the different factors affecting the binary code authorship attribution results reported in [8]?

*These authors contributed equally to the paper.

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–12
© YYYY Copyright held by the owner/author(s).
https://doi.org/XXXXXXX.XXXXXXX

RQ1 (Reproducibility Study). We first tried to reproduce the base results of Caliskan *et al.* [8], namely, the accuracy of author identification from 32-bit x86 executables, as the number of authors varies from 20 to 100. With 20 author datasets, we also studied how the results would hold for binaries compiled at different optimization levels (`-O1`, `-O2`, `-O3`, `-Os`) or stripped binaries (32-bit x86 with no optimizations enabled).

Overall, the accuracy numbers we obtained are considerably lower than those originally reported by Caliskan *et al.* [8].

(1) For the baseline experiments involving 100 authors using 32-bit unoptimized x86 binaries (i.e., compiled using `-O0`), Caliskan *et al.* report an accuracy of 96% [8] while we obtain an accuracy of 63%.
(2) For various different optimization levels (i.e., compiled using `-O1`, `-O2`, etc) Caliskan *et al.* report accuracies ranging from 89% to 96% for 100 authors while we obtain lower accuracies of around 80% for much smaller datasets involving 20 authors.
(3) For stripped binaries (32-bit x86, no optimization), Caliskan *et al.* report an accuracy of 72% for 100 authors while we obtain an accuracy of about 57% for 20 authors.

RQ2 (Cause Analysis). To understand the reasons for these differences in classification accuracy, we identified and analyzed the features that contribute most to author classification decisions.

(1) We found that data erroneously disassembled as code by the `ndisasm` disassembler used by Caliskan *et al.* played a significant role in their classification accuracy. The issue is that, given an executable file, `ndisasm` simply disassembles the entire byte stream for the file as code without any consideration of whether or not the bytes being disassembled come from a code region of the file. This results in non-code bytes from the file, such as section headers, string table, symbol table, etc., all being disassembled.
(2) To determine the extent to which our accuracy numbers were being inflated by such erroneous disassembly of data, we repeated our experiments on a 20-author dataset where `ndisasm`'s disassembly was limited to just the text section (the other tools in the pipeline were unaffected). For the 20-author dataset, the classification accuracy we initially obtained was 82%. However, limiting `ndisasm`'s disassembly to only the code bytes in the executables—and making no other change to any other part of the pipeline—resulted in a 14% drop in classification accuracy, from 82% to 68%. This indicates that the erroneous disassembly of data significantly impacts classification accuracy.

The reason is data bytes embedded in executable files can come from a variety of sources, such as string literals, numerical constants, and symbols, as well as metadata, such as headers and byte offsets referring to different points in the binary. Some of these sources may be unexpected: e.g., commonly used compilers such as GCC and Clang embed source filenames into binaries as metadata even when debugging is not enabled [36], which means that if source filenames in the experimental dataset incorporate author identifiers for convenience, these identifiers will appear in the symbol tables of unstripped binaries. Erroneous disassembly of such embedded data can then affect stylometry in unexpected ways. Such effects indicate the need for careful attention to toolchain-introduced effects, both for the tools used to prepare the binaries and those used to analyze them, to avoid inadvertently introducing errors into the stylometric analysis.

Research Artifacts: To promote further reproducibility and replication, we open-sourced the artifacts created, generated, and used in this research. These are available at: https://github.com/sprlab/binary-stylometry

Organization. The remainder of this paper is organized as follows. Section §2 provides background material, including a description of Caliskan *et al.*'s approach [8]. Section §3 discusses various challenges that can arise in binary-level code stylometry due to the effects of compilation and decompilation. Section §4 discusses the specifics of our research methodology, and Section §5 describes our reproducibility results, and Section §6 describes our in-depth study of finding the causes to explain the results. In Section §7, we discuss the implications of our findings, threats to validity, and lessons learned from this research. Finally, Section §8 discusses the related works, and Section §9 concludes the paper.

2 BACKGROUND

2.1 Structure of Executable Files

Stylometry of executable code relies fundamentally on parsing executable files to extract information about the code. It is, therefore, helpful to understand the structure of such files. Here, we briefly discuss the format of ELF (Executable and Linkable Format) binaries, which are used on Linux systems; other systems use executable formats that, while different in details, are conceptually very similar.

An ELF file consists of (1) an ELF header that describes properties of the file; (2) a Program Header Table that tells the operating system how create a process image when the file is executed; (3) a sequence of sections containing code and data; and (4) a section header table that describes these sections. Commonly occurring sections in GCC-generated ELF executables include [2]: `.text`, which contains the main body of executable code; `.data`, which contains initialized static variables; `.rodata`, which contains read-only data such as strings; `.plt` and `.got`, used for dynamic linking of library code during execution; `.symtab`, the symbol table; and `.strtab`, which contains strings associated with the symbol table, e.g., symbol names.

References from one part of an ELF file to another, e.g., from the ELF header to the section header table, from section header table entries to the corresponding sections, from instructions in the `.text` section to global variables in the `.data` section, etc., are typically given as offsets in the file. One implication of this is that changes to the size of one section of an ELF binary, e.g., due to optimization, can result in changes to the conceptually unrelated offsets embedded in other sections. This is significant because, as discussed later, erroneous disassembly of data bytes—including such offsets embedded in the binary—can significantly impact binary code stylometry results.

2.2 The Stylometry Pipeline of Caliskan *et al.*

The pipeline takes in a dataset of compiled binaries and produces a Random Forest Tree Classifier based on the features extracted

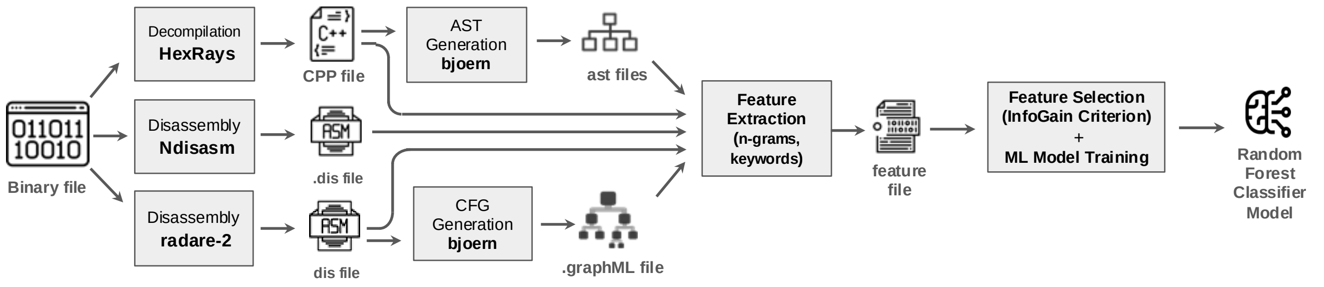


Figure 1: An overview of binary code stylometry pipeline used by Caliskan *et al.* [8]. It shows the process and tools used to extract *n-gram* features from (1) decompiled source codes, (2) dissembled code sequences, and (3) control flow graphs (CFGs). Next, after the dimensionality reduction of features through the Information Gain Criterion, random forest classifiers are trained.

from the dataset. Then, the classifier is evaluated on the test data set. The pipeline consists of the following 7 steps:

For the **training phase** of the ML pipeline the binaries of the dataset are processed via the following sequence of steps:

(1) **Decompilation.** In this step, the binaries are decompiled using IDAPro (version 8.3) for further processing.
(2) **Disassembly using ndisasm.** The binaries are disassembled using the *ndisasm* disassembler. Our experiments used version 2.15.05 of ndisasm.
(3) **Disassembly using radare2.** The binaries are disassembled using *bjoern-radare2*. Our experiments used radare2's version 5.7.9.
(4) **Abstract syntax tree (AST)-based feature generation using joern-tools.** AST-based features are generated from decompiled files using *joern-tools*. Our experiments used version 0.1 of *joern-tools*.
(5) **Control flow graph (CFG) generation using bjoern-radare2.** Control flow graphs are generated from disassembled files of the dataset using *radare2*. Our experiments used radare2's version 5.7.9.
(6) **Feature extraction and information gain criterion application.** In this step, all features from the extracted data sources are compiled into a single arff file. Subsequently, the information gain criterion is used to isolate the top 200 features with high information gain, which are then stored in another arff file.
(7) **Training the ML model.** A *Random Forest Tree* model from the Weka library (Java) is trained on the 200 extracted features. The trained model was then saved in a separate file for later use.

For the **testing phase** of the ML pipeline: Steps 1 to 5 are executed similarly to the training phase. In step 6, after gathering all features into one arff file, the same 200 features (selected during the training phase using the information gain criterion) were isolated and stored in a new arff file. These features are then tested against the previously saved model to obtain accuracy results.

3 BINARY-LEVEL CODE STYLOMETRY: ISSUES AND CHALLENGES

Binary-level stylometry seeks to determine code authorship from compiled binaries. In order to understand what factors could affect this process, it is useful to consider how such binaries are created and how authorship-relevant code characteristics might be obtained from them.

The process of compiling a set of source files pgm_1.c, …, pgm_n.c to an executable pgm.exe involves a number of steps. Each source file pgm_i.c is read in by the front end and transformed into an internal program representation (IR), which may optionally be optimized in various ways to improve performance, then translated to a relocatable binary pgm_i.o. The resulting relocatables are combined by a linker to generate the executable pgm.exe. The executable typically contains information about symbols in the program, e.g., function and variable names, which can be useful for debugging; interestingly, the executable can also contain metadata about the compilation process, e.g., GCC typically embeds the source file name and the name and version of the compiler into the executable as symbols. Such symbol information can be removed by a process called *stripping* that results in smaller files that are referred to as stripped binaries.

Code optimization can profoundly impact the structure of the code generated for a program [35]. This can change or obliterate code features that may potentially be useful for authorship attribution. This is illustrated in Figure 2, from [19].[1] In this example, the source code written by the programmer contains a function whose body contains a loop and a conditional. Code optimization eliminates all of the control flow in the original program, and the final executable contains only simple straight-line code that is very different from the structure of the original code. Conversely, optimizations aimed at reducing code size can combine multiple similar code fragments into "functions" invoked through function calls [12], resulting in binary-level control flow that is very different from that of the source code.

The identification of the possible authors of a binary requires extracting and analyzing code style characteristics from it. This

[1]This example uses C-like syntax (with added comments) for ease of understanding. In practice an optimizer would operate on an IR representation of the program.

```
void f(int m, int k) {
  for ( ; k > 0; k--) {
    if (k % 2 == 0) /* k even */
      m += 2*k;
    else   /* k odd */
      m -= 1;
  }
  printf("%d\n", m);
}

int main(int argc, char **argv) {
  int n = atoi(argv[1]);
  f(n, 3);
  return 0;
}
```

(*a*) Original program

```
int main(int argc, char **argv) {
  int n = atoi(argv[1]);

  int m = n;
  for (k = 3; k > 0; k--) {
    if (k % 2 == 0) /* k even */
      m += 2*k;
    else   /* k odd */
      m -= 1;
  }
  printf("%d\n", m);

  return 0;
}
```

(*b*) After function inlining

```
int main(int argc, char **argv) {
  int n = atoi(argv[1]);
  int m = n;
  int k = 3;

  if (k % 2 == 0)  /* iter 1 */
    m += 2*k;
  else
    m -= 1;
  k--;

  if (k % 2 == 0)  /* iter 2 */
    m += 2*k;
  else
    m -= 1;
  k--;

  if (k % 2 == 0)  /* iter 3 */
    m += 2*k;
  else
    m -= 1;
  k--;

  printf("%d\n", m);
  return 0;
}
```

(*c*) After loop unrolling

```
int main(int argc, char **argv)
{
  int n = atoi(argv[1]);
  int m = n;

  m -= 1;  /* iter 1 */
  m += 4;  /* iter 2 */
  m -= 1;  /* iter 3 */

  printf("%d\n", m);
  return 0;
}
```

(*d*) After constant folding and dead code elimination

Figure 2: An example of the effect of compiler optimizations (from Jacobsen *et al.*, 2021 [19])

```
#include <stdio.h>
long w = 0x1234567890abcdef;

int main() {
  printf("%d\n", (int)w);
  return 0;
}
```

```
00ef        add bh,ch
cdab        int 0xab
90          nop
7856        js 0x306c
3412        xor al,0x12
```

Figure 3: An example of erroneous disassembly in ndisasm. When the executable for the program on the left is disassembled, the initialization constant `0x1234567890abcdef` of the variable `w` is disassembled as code, resulting in the spurious instructions shown on the right (each spurious instruction is shown preceded by its binary encoding). The order of bytes in the disassembly is reversed compared to the source code because the processor used was little-endian.

typically involves examining the binary code itself, e.g., by disassembling the machine code to assembly code that is human-readable and more amenable to processing, and possibly decompiling the disassembled code to a higher-level representation such as source code. Disassembly issues can affect authorship-relevant code features in two ways. The first arises from the fact that distinguishing between instructions and embedded data in a program's code region is undecidable in general, and disassemblers typically resort to heuristics that can sometimes result in disassembly errors [41]. The second arises from disassembly of non-code memory regions. Figure 3 shows an example of the second kind of error from the `ndisasm` disassembler used in Caliskan *et al.*'s work. In each case, the resulting disassembled code does not accurately reflect the actual program code and thus can distort the features used for stylometry.

There are three takeaways here. First: compiler optimizations can profoundly affect the structure of the generated code, and thereby influence stylometry-relevant features extracted from it. Second: disassembly—the first step in any analysis of binary code—can introduce its own inaccuracies and further blur and distort such features. Third: the effects of compilation and decompilation can vary depending on program characteristics, the compiler used and compiler options selected, and the disassembler/decompiler used.

4 METHODOLOGY

4.1 Data Selection

The specific dataset used by Caliskan et al. [8] was not available due to privacy considerations, so we followed their description of dataset construction as closely as possible to create a dataset that we believe is similar to theirs. In particular, we followed Caliskan *et al.* [8] in using Google Code Jam submissions [2], and used the same criteria as in their paper to select authors and their submitted code samples. While our resulting dataset is not identical to that of Caliskan *et al.*, we believe they are similar–the only differences between their dataset and ours are the following: a) they used GCJ submissions from 2008 to 2014 while we used GCJ submissions from 2008 to 2020, and b) the tasks and authors we selected may differ from the original dataset. [3]

We extracted all files available in the repository and filtered for C++ submissions to align with Caliskan et al.'s work. We mapped all unique tasks to all unique authors who submitted to them and used this mapping to identify the maximum number of authors that

[2]We fetched the Google Code Jam submissions from this public repository: `https://github.com/Jur1cek/gcj-dataset.`
[3]We also experimented with GCJ submissions from 2008 to 2014, but found that the resulting dataset showed a notable drop in accuracy for a small sample subset. For these reasons, we focused our experiments with the dataset from 2008-2020.

submitted to same 9 task combinations.[4] We ended up with 801 authors who submitted to the 9 GCJ tasks. There were two submissions per task in the dataset. We selected the first submission. To understand the difference between two submissions, we computed the average normalized edit distance between the two submissions of all 801 authors and found that it was 0.02[5], suggesting that both submissions were almost identical. Upon some manual inspection, we found that submissions differed in the following ways: a) usage of larger data types in the second submission (e.g., long instead of int) and b) usage of different static paths to load the dataset.

4.2 Data Processing

To ensure fidelity when compiling these programs to executables and extracting features from the executables, we used the same tools as in their paper; where possible, we used information from the GitHub repository specified in the paper to use the same versions of the tools and invoked them with the same command-line arguments. Specifically, we used the repository associated with Caliskan's work [1] to process the datasets. We developed wrapper scripts around the code to adapt it to a new environment, and the code was containerized using Docker to parallelize the pipeline. The detailed description of the pipeline is given in Section 2.2. We also performed some preprocessing to align the pipeline with the dataset. For example, we preprocessed the GCJ dataset by removing the dot character ('.') from the author names, as this was a requirement for the pipeline to operate correctly. Initially, only 319 authors had a successful compilation using the `g++-4.8` compiler. We added the `-std=c++0x` flag to the compilation command, setting the standard to C++11, which enabled the successful compilation of 647 authors. We then discarded the authors who had any compilation errors. For baseline unoptimized x86 compiled binaries, table 1 shows that 647 out of 801 authors remained after discarding due to compilation errors. It is worth noting that this successfully compiled author count is more than enough for our experimentation.

Number of authors who submitted to all 9 tasks	801
Compiled Successfully	647

Table 1: The number of authors remaining after each step for the 2008 to 2020 GCJ submissions.

Using IDAPro (version 8.3) as the decompiler, we decompiled the binaries by using the following arguments:

`-Ohexrays:{`*decompiledFile*`}:ALL -A {`*inputFile*`}`

This configuration decompiles all the functions in the code section of the binary and dumps the decompiled code in the specified output file. We used 9-fold cross-validation to create each dataset. We did an 8/1 split of the 9 tasks for each author in the dataset in each validation round; the training set contained 8 problems per author, and the testing set contained 1 problem per author.

In respective sections, we discuss how we used this dataset to answer our research questions.

[4]This is because Caliskan *et al.* use 9 common samples for each author.

[5]A distance of 0 means no difference at all, whereas a distance of 1 means complete difference.

4.3 Performance Metrics

All the accuracies reported in this paper are an average score across n different datasets. For different experiments, this number is described further in Section 5.1. Following Caliskan *et al.* [8], for each dataset, we calculated the accuracy over *k-fold* evaluation. Folds refer to the 9 possible train-test splits of the tasks. Classification accuracies are computed as follows. The accuracy for a single fold is given by:

$$Af_i = \frac{\text{number of correctly classified authors}}{\text{total number of all authors}}.$$

Accuracy across k folds (in our case, $k = 9$) is computed as follows, where Af_i represents the accuracy of the model on the i-th fold:

$$\text{Accuracy} = \frac{1}{k}\sum_{i=1}^{k} Af_i$$

The average accuracy across n datasets is given by:

$$\text{Average Accuracy} = \frac{1}{n}\sum_{i=1}^{n} A_i$$

where A_i represents the accuracy of the i-th dataset. This final formula represents the accuracy used for results and Figures below.

5 RQ1: REPRODUCIBILITY STUDY

The first part of our reproducibility study focuses on a collection of unoptimized unstripped 32-bit executables for the Intel x86 architecture that were obtained using the g++ compiler on a set of C++ code submitted to Google Code Jam. It forms the core of the work for reproducing [8]. The next part of our reproducibility study considers two settings, i.e., the use of *i)* compiler optimizations, *ii)* stripped binaries as originally considered by Caliskan *et al.* [8]. Here, we specifically answer the following research questions:

- RQ1.1: To what extent authorship attribution results from [8] are reproducible for regular Intel x86 binaries without code optimization?
- RQ1.2: To what extent authorship attribution results from [8] are replicable for Intel x86 binaries with (1) different code optimizations enabled and (2) symbols stripped?

5.1 Experimental Setup

For all of our experiments presented in this work, we used 3 Machines with the following configurations:

- 2 × 6-core Intel® Xeon® CPU X5650 @ 2.67GHz
- 48 GB RAM

Next, we discuss experimental setups to answer each sub-RQ for our reproducibility study.

Setup for Regular x86 binaries. We follow Caliskan *et al.*'s compilation settings and compile all source code with g++4.8, -m32, and -O0 flags for x86 architecture. We created 20, 50, and 100-author datasets. For each n-author dataset, we sampled n-authors randomly with replacements from the pool of 647 authors. We create 5, 5, and 4 datasets for 20, 50, and 100 authors, respectively. The processing times to compute the results for the regular x86 datasets were as follows: approximately 2 days for the 20-author dataset, 10 days

Author Count	Accuracy Caliskan et al.	Accuracy ± Std Dev (Our Work)
20	99.0	81.55 ± 7.56
50	-	73.20 ± 4.60
100	96.0	62.89 ± 1.60

Table 2: Reproducibility of classification accuracy: unoptimized unstripped 32-bit x86 binaries. A hyphen ('-') denotes that the corresponding value is unavailable.

for the 50-author dataset, and 20 days for the 100-author dataset. Since the runtime grows exponentially as the number of authors increases, we chose 4 instead of 5 for the 100-authors experiment.

Setup for Optimized and Stripped Binaries. To study the impact of optimization, we experiment gcc optimization flags `-O1`, `-O2`, `-O3`, and `-Os`. Although the original work did not report results for `-Os`, we included it for the sake of completeness. For this study, we chose the 20-author count to ensure a meaningful comparison with baseline experiment results (RQ1.1), where we observed the best performance on average. We compile all the code for the 32-bit x86 architecture. The compilation commands used are:

`g++-4.8 -m32` *optflag* where *optflag* ∈ {`-O1`, `-O2`, `-O3`, `-Os`}.

To replicate the results for strip binaries, we strip the executables by adding the `-s` flag to the compilation command.

5.2 Results

Performance on Regular x86 Binaries. Table 2 shows our results for baseline author classification accuracy on unstripped unoptimized 32-bit x86 binaries. There are two key takeaways here.

(1) There is considerable variability in classification accuracy for different experiments. For example, for 20 authors, accuracy ranges from a low of 70.0% to a high of 89.4%. This indicates that the accuracy of binary-level code stylometry is sensitive to code characteristics.
(2) The author classification accuracy numbers we obtain are significantly lower than those originally reported by Caliskan *et al.* [8]. For example, Caliskan *et al.* report an accuracy of 96% for 100 authors while we obtain an average accuracy of only 63%. It is possible that some of this difference in accuracy may be due to differences in the code samples used (see the previous point; recall that our dataset is similar to, but not identical with, that used by Caliskan *et al.*). However, our investigations suggest that toolchain-based effects, discussed in Section 7, may also be a factor.

Performance on Code Optimization. Table 3 shows the classification accuracy for different levels of code optimization in the column labeled "Accuracy (baseline)".

For optimization levels compiled using `-O1`, `-O2`, and `-O3`, Caliskan *et al.* report accuracies ranging from 89% to 96% for 100-authors. In our baseline experiments of 20-author datasets, our accuracy results for different optimization levels range in a narrow band between 79% and 81%. Our in-depth cause analysis in Section §6.2, reveals

Optimization	Accuracy (baseline)	Accuracy (.text section only disassembly)
-O0	81.55	68.22
-O1	80.66	57.44
-O2	79.11	54.22
-O3	80.11	54.00
-Os	80.22	56.22

Table 3: Performance comparison of different disassembly methods using `ndisasm` on 20-author datasets: RQ1.1 (unoptimized 32-bit binaries) and RQ1.2 (optimized 32-bit binaries). 'Baseline' means that the original disassembly method from Caliskan's work was used.

that features obtained from disassembling code with `ndisasm` actually contain non-code features, too. This is because `ndisasm` erroneously considers the whole binary a continuous stream of code bytes – thus, it disassembles all the data, metadata, and symbols as code. We hypothesize that the minimal effect of code optimization may be due, at least in part, to the influence of dissembling non-code sections. The reason is code optimizations do not change the non-code sections and, therefore, do not significantly impact features obtained from these sections. To test this hypothesis, we retrained the models by keeping only the `ndisasm` disassembly output of the `.text` (code-based) section. For this experiment, we found that our accuracy results for different optimization levels range between 68% and 57%. We see a significant drop in performance after restricting the `ndisasm` disassembly to the executable code section only. This degrade lends support to our preceding hypothesis. We observe that optimization with the `-O2` and `-O3` flag results in the highest accuracy drop with ∼14% when compared with `-O0` flag dataset.

Performance on Stripped binaries. Stripping a binary removes symbol information from it; the code remains unchanged.

In our experiments, we get an accuracy of 57.74%. Hence, classification accuracy for the 20-author datasets drops by about 24% for stripped unoptimized binaries compared to unstripped ones. This drop is consistent with the results of Caliskan *et al.* In Section §6, we observe that most of the important features for classification are derived from the erroneous disassembly of non-code sections, including the symbol table. We hypothesize that the large drop in classification accuracy resulting from the removal of symbol information occurs in significant part due to the consequent absence of disassembly of symbol bytes.

6 RQ2: CAUSE ANALYSIS

To understand the factors affecting classification accuracy, we performed an explanability analysis of a subset of our results.

6.1 Methodology

We followed the steps described below.

Model Selection. A model in this context means the Random Forest Classifier. We picked the models corresponding to the 20-author count datasets from RQ1 as this is sufficient to provide us with important insights. There were five datasets in the 20-author count. To have a better understanding of the overall performance, we considered the best-performing and worst-performing datasets. Within these two datasets, there were nine different models for each fold. We selected the top fold model from the best dataset and the bottom fold model (in terms of classification accuracy) from the worst dataset, respectively.

Author Selection. We picked the top 5 correctly classified authors (in terms of confidence score) from the top fold model and the bottom 5 misclassified authors from the bottom fold model.

Feature Selection. The Random Forest Classifier (RFC) Model used in the pipeline is built from 200 features. We used a subset of these features for our analysis.

We used SHAP (SHapley Additive exPlanations) Framework to analyze features [26]. SHAP is a commonly used framework to aid the explainability of machine learning models. SHAP framework computes 'Shapley' values that are linked to each feature in the model. These Shapley values determine the relative importance of features in predicting the output. We used SHAP's Python implementation for our processing. Within the SHAP framework, we used a black-box explainer called the Kernal SHAP [25]. Kernal SHAP is a game theoretic technique that is based on weighted linear regression to estimate the importance of each feature [25].

Since the RFC model is implemented in Weka [16], which is a Java library, we performed the following steps to link the RFC model with SHAP's Python framework. We extracted the text version of the Random Forest Tree from Weka. Then, we parsed the RFC model to implement it in a custom Python object. This object implemented a prediction function that generated a classification score for each class given an input data point. To ensure fidelity, we cross-checked our function's output against the original output from the Weka library. This prediction function was provided as an input to the SHAP's Kernel explainer. We computed Shapley values per class: a class in our context refers to an author. Therefore, we computed Shapley values of features per author.

High Shapley values indicate that a feature is more important for a given author, while low values indicate the opposite. We selected the top 10 features (in terms of Shapley values) of the selected authors. We also computed the corresponding Attribute Importance values (by the Mean Decrease in Impurity method) of such features to assess their overall impact on the model. The mean Decrease in Impurity (MDI) of a feature is defined as the weighted impurity decrease of the nodes in which a feature is used in all the trees of the Random Forest [24]. A high MDI value indicates high relevance of the feature in the model, whereas a low value indicates the opposite[7, 24]. Weka library allows the computation of these Attribute Importance values. Please note that this importance value provides a measure of feature importance across all authors, whereas Shapley values are importance values of features per author.

Analysis Method. We then analyzed the selected features to understand how these features contribute to the model.

Three types of features appear in top-10 features: `ndisasm`, AST, and `bjoern`. The `ndisasm`-based features are n-grams derived from the disassembly of a given program binary, generated using `ndisasm`. Similarly, `radare2`-based features are n-grams from the disassembly produced by `radare2` for the same program binary. The AST-node-based features are derived from the abstract syntax tree (AST) of the decompiled C++ file obtained through IDAPro. To analyze AST-based features, we looked at the line and constructed in the decompiled C++ file from which the AST node originated.

To analyze `ndisasm`-based features, we followed these steps: First, we matched each feature's text with the text in the `ndisasm` disassembly to identify the corresponding instructions from which the feature text originated. Then, we retained the machine code of these identified instructions. Next, we located the section in which this machine code occurred. Finally, we classified each feature based on its section of origin as either code-based or non-code-based. If a feature appeared in both code-based and non-code-based sections, it was classified as code-based.

To analyze `radare2`-based features, we followed a similar approach: we matched the feature text with the `radare2` disassembly to identify the corresponding instructions. We then examined the comments surrounding each feature to detect common patterns. `radare2` adds comments to instructions, such as function headers, whenever a function is called within an instruction.

6.2 Findings

Our findings are structured as follows: we first discuss the key characteristics of the dataset and discuss the top features, then examine how specific features contribute to authorship attribution.

Dataset and feature characteristics

We considered the features of the best and worst-performing datasets of 20-author count. The best-performing dataset has a classification accuracy of 89%, while the worst-performing dataset has a classification accuracy of 70%. The top feature sources of the corresponding models are listed in Table 4. Their types are further elaborated in Table 5. In Table 5, we observe that most `ndisasm`-based features are largely dictated by line bigrams. In Table 4, we observe that the top features have very low Shapley values (0.001 to 0.01) but relatively high importance (Mean Decrease in Impurity) values (0.5 to 0.8). There can be various reasons behind low Shapley values [10], including the presence of redundant signals or the model relying on a broad range of features rather than a single dominant one. High attribute importance values indicate that these features play a role in discriminating multiple authors. Next, we discuss the important findings below.

Erroneous disassembly leads to erroneous features

`ndisasm` based features are the most frequent sources of the top features of the top authors, as observed in Table 4 and. A problem here is that `ndisasm` disassembles all of the bytes of the input binary to instruction code without making any distinction between bytes that occur in the code or non-code sections. We found out that 29 out of 33 `ndisasm` based features resulted from erroneous disassembly of machine code bytes from non-code based sections (e.g., .strtab) to instruction code. To understand erroneous disassembly in the model, we analyzed ASCII representations of the corresponding machine code of features and categorized common patterns. Table 6 shows the classification, with details provided below.

Feature Sources	Unique Features (64 out of 100)			SHAP value ranges	Attribute Importance Range
	High performing Authors Only	Low performing Authors Only	Common		
ndisasm Disassembly	17	16	6	0.001-0.01	0.6-0.8
radare2 Disassembly	16	8	2	0.001-0.005	0.5-0.8
Decompiled code	2	5	0	0.002-0.007	0.6-0.8

Table 4: Features descriptions of top and low performing datasets for the selected authors of top and low performing datasets for the selected authors.

Feature Type	ndisasm	radare2	Decompiled Code
Instruction Unigrams	1	4	-
Instruction Bigrams	3	8	-
Instruction Trigrams	-	6	-
Line Unigrams	-	2	-
Line Bigrams	29	4	-
AST Node Avg Dep	-	-	2
AST Node TF	-	-	1
AST Node TF-IDF	-	-	4
Total	33	24	7

Table 5: Feature types for the top and low-performing datasets across selected authors.

Feature Type	Freq. of Occurrence
Source Filename	6
Repetitive Strings	2
Function strings as symbols	4
Non-alphanumeric Bytes	6
Miscellaneous Symbols Strings	13

Table 6: Different kinds of erroneous disassembly features from `ndisasm` picked by the model. The frequencies (not mutually exclusive) are counted out of 29 such erroneous features.

- `Source Filename`: This feature correspond to source filenames. This is because *g++* compiler embeds the filename of the source CPP file into the binary. An example string is "'58808576_timics23.cpp". This string contains the algorithmic problem identifier and the author's username. This is problematic because the label (author username) encoded as a feature is being used as a feature to classify the class (author username). This indicates the contamination of train-test data.
- `Repetitive Strings`: This feature corresponds to repetitive strings used by the authors, located in the binary's data section. An example string is "0.out".
- `Function strings symbols`: This feature corresponds to the function name used in the binary that is embedded as a symbol string by the *g++* compiler. An example string is "`freopen`".
- `Non-Alphanumeric Bytes`: These features correspond to byte sequences that do not translate to any alphanumeric strings. Examples include file offsets embedded in the executable (see Section 2.1).
- `Miscellaneous Symbol Strings`: These features correspond to miscellaneous symbol strings found in the binary, originating from various non-code sections, including the .strtab section. An example symbol string is "endlIcSt11".

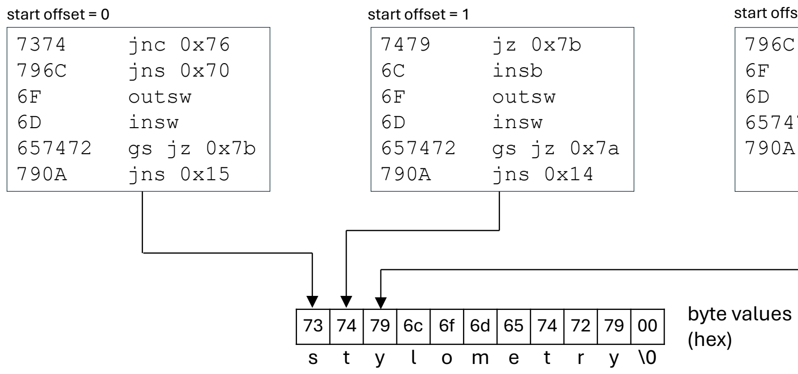
Erroneous features impact classification accuracy
Due to our observation that the majority of `ndisasm`-based features (29 out of 33) are the result of erroneous disassembly of non-code sections, we conducted the following experiment on the 20 author datasets, we used for RQ1: we replaced `ndisasm` disassembly output for all sections in the binaries with the disassembly output for only the code section (`.text`). This would have the effect of removing erroneous disassembly-based features from the model, as we only included the disassembly of the code-based section. We then reran the pipeline to get the accuracy results. The resulting accuracies are shown in Table 3. As can be seen in the table, we observed a marked decrease in accuracy for both unoptimized and optimized datasets. This observation lends support to the hypothesis that erroneous features derived from `ndisasm` are skewing the performance.

Duplication of features
Among the top features, 4 out of 16 radare2 disassembly-based features and 2 out of 7 AST-based features correspond to the library function `scanf`. In radare2 disassembly, features are derived from the text in radare2's disassembly comments, which include the names of functions called within instructions. Additionally, 2 erroneously disassembled features from `ndisasm` also correspond to the usage of `scanf`. This implies that the pipeline is susceptible to duplication of signals.

7 DISCUSSION

7.1 Stylometric Implications of Disassembling Data as Code

As discussed in Section 6.2, our experiments indicate that erroneous disassembly of data, such as error messages and format strings (from the `.rodata` section of the binary), initialized global data (from the


start offset = 0

```
7374     jnc 0x76
796C     jns 0x70
6F       outsw
6D       insw
657472   gs jz 0x7b
790A     jns 0x15
```

start offset = 1

```
7479     jz 0x7b
6C       insb
6F       outsw
6D       insw
657472   gs jz 0x7a
790A     jns 0x14
```

start offset = 2

```
796C     jns 0x6e
6F       outsw
6D       insw
657472   gs jz 0x79
790A     jns 0x13
```

Figure 4: An example of disassembly differences arising from different instruction start offsets within a string ("stylometry")

`.data` section), and symbols (from the `.symtab` and `.strtab` sections) significantly impact the results of stylometric analysis. It can be tempting to argue that since strings (in `.rodata`) and function names (in `.symtab` and `.strtab`) can be reflective of programming style, disassembly of such data as code can indirectly but nevertheless usefully recover stylistic characteristics from the binary. This argument is flawed, however, for two reasons:

(1) On variable-instruction-length architectures such as the x86, the "code" obtained from the disassembly of strings and symbols is strongly dependent on the bytes that happen to occur at locations the disassembler considers to be instruction start bytes. The particular bytes within a string that are interpreted as instruction start bytes depend on how the earlier bytes were decoded, which in turn depends on the size and contents of the preceding bytes.[6] This means that small changes to the bytes preceding a string, e.g., due to an optimization-induced change in the size of an earlier section or an offset in the code, can cause the instruction start offsets within that string to change, which can then result in a different disassembled instruction sequence. This is illustrated in Figure 4, which shows how disassembling a string starting at different offsets within the string (in this example, "`stylometry`") produces different disassembled code sequences.

(2) An executable binary contains many different types of data, most of which have little to do with the code author's programming style. Header sections contain file metadata, e.g., section offsets. In addition to strings, the `.rodata` section contains other read-only data, such as jump table offsets, that have nothing to do with style. Sections such as `.got` (Global Offset Table) contains information for use during dynamic linking. Treating such data as code risks injecting meaningless noise into the stylometric analysis.

For these reasons, a more robust alternative to disassembling the entire contents of the executable file as code (as `ndisasm` does) would be to limit disassembly to code sections and extract strings and/or symbol names as features separate from the disassembled code.

[6]When decoding a byte sequence during disassembly, the first few bytes provide information about the length of the instruction being decoded, which then determines the start byte for the next instruction.

7.2 Threats to Validity

Dataset Differences

Like the original work, our experiments are based on the Google Code Jam (GCJ) dataset. GCJ dataset contains algorithmic tasks that are solved by the participating programmers (authors). The authors and the tasks that we selected from the dataset may differ from the original work. We selected the portion of the dataset from the years 2008-2020, whereas the original authors used the portion of the dataset from the years 2008-2014. Note that we initially sampled the GCJ dataset from 2008-2014. However, we were getting poor accuracy results. For five 20 author datasets and three 50 author datasets, we found the average accuracy to be around 58% and 40% respectively. Due to this, we sampled the dataset from the years 2008-2020, for which the accuracy was closer to Caliskan's work.

For splitting the dataset (train-test split), we followed the methodology of Caliskan *et al.*'s work. Given that we followed Caliskan *et al.*'s methodology in building the dataset, the difference described earlier should not impact our broader findings.

Toolchain versions

We used certain tools that may have newer versions compared to the ones used in the original pipeline. The decompiler in the pipeline is HexRays from IDA Pro. We used version 8.3 of IDA Pro, which was released after the publication of the original work. This may introduce slight differences in the output produced but should not impact our broader findings. Please look at Section§2.2 for all toolchain versions used.

Pipeline Differences

We inherited the pipeline given in the repository linked with the original work. We followed the readme file and used the scripts listed in it to build the new pipeline. In the original repository, the code needed certain modifications to run in a new environment (i.e., certain paths were hardcoded). To adapt it to our environment and toolchain, we made changes to the code. We made efforts to ensure that the modified pipeline remains as close to the original as possible. Specifically, one researcher built the pipeline, whereas two other researchers independently checked the compliance of the pipeline with the original codebase released by Caliskan *et al.* [8].

For dimensionality reduction, the original paper mentions that the extracted features were reduced by two dimensionality reduction approaches: a) Information Gain Criterion and b) correlation based feature selection. We applied the first approach, as it was the only approach available in the corresponding source code repository. The original paper reduced the total features to 53 after dimensionality reduction, but we found the performance with 53 features to be low. Thus we chose to reduce the feature set to 200 instead as it produced better performance.

7.3 Lessons Learned

Our experimental results and their analysis point to some lessons that we believe are important for research on and applications of binary code stylometry.

Toolchain artifacts can skew results.
Our experiments demonstrated two ways in which stylometric results could be affected by artifacts inadvertently introduced by the analysis toolchain: improper disassembly of data sections in the binary and source filenames embedded into the binary as symbols by the compiler. It is possible that there are other such toolchain artifacts that we did not detect. For stylometry work, this argues for (1) careful preparation of datasets and analysis tools ahead of time to guard against such artifacts; and (2) careful after-the-fact analysis of the results to check that they have not been affected by unexpected features. (On the latter point, it should be noted that determining whether a feature is "unexpected" can be subtle and require considerable domain expertise. In the course of this work, for example, we were intrigued by the observation that many highly ranked features used instructions, such as '`arpl`', '`insb`' and '`outsd`', which in our experience are rare in application code. Identifying where these instructions originated involved tracing through the structure of ELF binaries, i.e., determining the offsets of these instructions in the binary and mapping the offsets to the sections they came from. This led us to the realization that they originated from the binaries' symbol table sections.)

Classification accuracy is sensitive to code characteristics.
Our results indicate that there can be a high degree of variability in the accuracy of author attribution. For example, for the 20-author results shown in Table 2, the average accuracy is 81%, but for individual datasets, it can be as low as 70%. An individual fold accuracy of the same dataset can be even lower (e.g., 50%). From an application perspective, this implies that average accuracy is not a good indicator of how confident we can be in the classification accuracy for a particular code sample.

Misattribution can cause harm.
Much of the work on binary-level code stylometry has focused on its ability to recover authorship-relevant information from binaries. For example, for 100-author stripped unoptimized 32-bit binaries, Caliskan *et al.* observe an average classification accuracy of about 72%[7] [8] and conclude that "stripping symbol information from executable binaries is not effective enough to anonymize an executable binary sample." Meng obtains an average accuracy of about 52% for 282 authors and concludes that "authorship identification is practical at the basic block level" [29]. In each case, the authors correctly point out that, from the perspective of extracting authorship-relevant information from binaries, the accuracy results obtained are significantly better than random chance.

[7] In the paper, this value is given as "a decrease . . . by 24%" from the 96% accuracy they observe for unstripped binaries.

From a privacy perspective, however, these results are worrisome. An average accuracy of 75% means that misattribution occurs 1/4 of the time on average; an accuracy of 52% means that attribution is wrong almost half the time. Furthermore, as pointed out above, the accuracy on any individual dataset may be considerably worse than the average accuracy. This can be a problem, because—regardless of the intent behind the stylometric analysis, e.g., identifying malware authors or unmasking protesters as described in Section 1—attribution errors can cause harm to anyone who is incorrectly identified as an author of the software under analysis (especially if the stylometry results are interpreted by authorities who may not be attuned to such nuances). The potential for misattribution should be understood and taken into consideration in any application of binary code stylometry.

For these reasons, we believe that evaluating stylometry results on binary code using only average accuracy values gives an incomplete picture. At a minimum, the variance should also be mentioned for context, together with a clear description of characteristics of the datasets used to obtain those results.

8 RELATED WORK

Most of the research work on code stylometry has focused on source code, though recent years have seen a growing interest on stylometry for executable binaries.

8.1 Binary code stylometry

The earliest work on binary-level authorship attribution that we are aware of is that of Rosenblum *et al.* [38]. Like the work of Caliskan *et al.* [8], this work assumes that each program has a single author. It uses clustering based on features derived from the control flow graph of the binary together with byte n-grams, instruction idioms, and information about library calls. They evaluate their approach using Google Code Jam data from 2009 and 2010 together with code written by undergradate students for an operating systems course; specifics of the binary creation process, such as machine architecture, compiler, optimization level, and stripping, are not discussed. The accuracy results reported are roughly comparable to ours: 81% for 10 authors and 51% for 200 authors.

Meng considers the problem of binary-level authorship attribution under the more realistic assumption that the binary may have multiple authors [29, 30]. Unlike most other work on this topic, Meng focuses on authorship attribution at the level of individual basic blocks, and accordingly uses code features derived from basic block characteristics. On a collection of 831K basic blocks from 170 binaries with 282 authors, obtained using `gcc` with `-O2` optimization, this work achieves 52% average accuracy at attributing authorship of individual basic blocks.

The work most closely related to ours is the binary-level stylometry work of Caliskan *et al.* [8]: the work described here aims to replicate their core findings. The technical details of their approach as well as their main results have already been discussed earlier, so we do not repeat them here. In addition to the core results discussed earlier, Caliskan *et al.* also apply their ideas to other settings, including obfuscated binaries and malware, which are orthogonal to the focus of our work and therefore not considered for this replication study.

Abuhamad *et al.* describe a deep-learning-based approach to code stylometry that is applicable at both the source code and binary levels [3]. Their approach to binary code stylometry is to decompile the binaries and apply deep learning to the resulting decompiled code. The authors report very high classification accuracies for a Google CodeJam dataset, including for binaries compiled with different levels of code optimization, stripped binaries, and obfuscated binaries. Their results are intriguing given that optimization and/or obfuscation can change code characteristics profoundly (e.g., see Figure 2), and it is conceivable that, for the optimized and obfuscated binaries used in their work, authorship clues may have been derived from data such as error messages and format strings rather than code, since such data are typically not affected by optimization/obfuscation. However, the paper does not discuss which code features recovered from decompiled binaries are relevant for stylometric classification, and the code and datasets used for this work are unavailable, so we were unable to explore this issue further.

8.2 Source code stylometry

There is a considerable body of work on source-level code stylometry, including both "regular" approaches aimed at identifying authors [3, 5, 17, 22, 40, 44, 45] and adversarial approaches aimed at hindering such identification [19, 34]. Source-level stylometry has two advantages compared to that at the binary level: (1) source code is typically much richer in features that are strongly associated with individual programmer style, e.g., comments, variable naming conventions, and indentation style; and (2) these features are available as written by the programmer without the profound transformations that can be effected by a compiler (see Section 3; however, it should be noted that such features may be attenuated by adherence to organizational style guidelines and/or the use of code formatters such as GNU `indent`). Since source-level stylometry is orthogonal to the focus of our work, we do not discuss it further.

8.3 Reproducibility and Replicability of Results

Reproducibility and replicability studies are important to locate errors, biases, and drift in scientific evaluation. Most of the reproducibility studies in security, internet, and software engineering measurement show different results than the originally reported [11, 13, 28]. Existing studies identified that the main challenges towards reproducibility are underspecified scientific methodology or the lack of precise description of the experimental setup [13, 14, 27, 28]. For example, Demir *et al.* [13] noted that slight differences in the experimental setup directly affect the overall results in the context of web measurement.

On the other hand, the leading cause of scientific studies falling short on "replicability" is a mismatch between experimental and real-world setups [6, 18, 31, 39, 42]. Arp *et al.* [6] showed that this mismatch could lead to a wide range of problems in the context of building machine learning-based security applications. Authors also noted that the Google CodeJam dataset used by most of the code authorship attribution works might not ideally represent real-world setting [6]. This is because most of the authors in this dataset tend to copy code snippets across multiple files, which might lead to inflated results. This work explores the unique challenges for authorship attribution of binary codes and studies their impacts.

9 CONCLUSION

We performed a replication study of influential recent work on binary-level code stylometry by Caliskan *et al.* [8]. We found the following important findings. The performance of binary code authorship attribution, we found, is significantly lower than reported in the original work (for 100 authors, we achieved approximately 63% accuracy, compared to the 96% reported in the original work). Secondly, an analysis of the features used in the pipeline indicates a significant number of features were a result of erroneous disassembly. Specifically, we found 29/33 of top `ndisasm`-based features were the result of erroneous disassembly. We found that this caused the model to use spurious features, i.e., the original file name, as the g++ compiler embeds the filename of the source CPP file into the binary – which might inflate the results.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under award no. CCF-2217215. Aylin Caliskan provided helpful feedback on an earlier draft of the paper.

The authors used Chatgpt4-o to refine the grammar and readability of Section §6.

REFERENCES

[1] 2015. calaylin/bda. https://github.com/calaylin/bda.
[2] 2020. https://docs.oracle.com/cd/E53394_01/html/E54833/elf-23207.html
[3] Mohammed Abuhamad, Tamer Abuhmed, David Mohaisen, and Daehun Nyang. 2021. Large-scale and robust code authorship identification with deep feature learning. *ACM Transactions on Privacy and Security (TOPS)* 24, 4 (2021), 1–35.
[4] Saed Alrabaee, Paria Shirani, Mourad Debbabi, and Lingyu Wang. 2017. On the feasibility of malware authorship attribution. In *Foundations and Practice of Security: 9th International Symposium, FPS 2016, Québec City, QC, Canada, October 24-25, 2016, Revised Selected Papers 9*. Springer, 256–272.
[5] Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. 2017. Source code authorship attribution using long short-term memory based networks. In *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I 22*. Springer, 65–82.
[6] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and Don'ts of Machine Learning in Computer Security. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 3971–3988.
[7] Leo Breiman. 2001. *Machine Learning* 45, 1 (2001), 5–32. https://doi.org/10.1023/a:1010933404324
[8] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. 2018. When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries. In *Proceedings 2018 Network and Distributed System Security Symposium (NDSS 2018)*. Internet Society. https://doi.org/10.14722/ndss.2018.23304
[9] Jean-Baptiste Camps, Christelle Chaillou, Viola Mariotti, and Federico Saviotti. 2022. Textual, Metrical and Musical Stylometry of the Trouvères Songs. In *Digital Humanities 2022 (DH2022)*.
[10] Molnar Christoph. 2020. Interpretable machine learning: A guide for making black box models explainable. (2020).
[11] Nadia Daoudi, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. 2021. Lessons Learnt on Reproducibility in Machine Learning Based Android Malware Detection. *Empir. Softw. Eng.* 26, 4 (2021), 74.

[12] Saumya K Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler techniques for code compaction. *ACM Transactions on Programming languages and Systems (TOPLAS)* 22, 2 (2000), 378–415.

[13] Nurullah Demir, Matteo Große-Kampmann, Tobias Urban, Christian Wressnegger, Thorsten Holz, and Norbert Pohlmann. 2022. Reproducibility and Replicability of Web Measurement Studies. In *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*, Frédérique Laforest, Raphaël Troncy, Elena Simperl, Deepak Agarwal, Aristides Gionis, Ivan Herman, and Lionel Médini (Eds.). ACM, 533–544.

[14] Jesús M. González-Barahona and Gregorio Robles. 2012. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empir. Softw. Eng.* 17, 1-2 (2012), 75–89.

[15] L Gurak, John Logie, M McCaughey, and MD Ayers. 2003. Cyberactivism: Online activism in theory and practice.

[16] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 10–18. https://doi.org/10.1145/1656274.1656278

[17] Aylin Caliskan Islam, Richard E. Harang, Andrew Liu, Arvind Narayanan, Clare R. Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing Programmers via Code Stylometry. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 255–270.

[18] Arthur Selle Jacobs, Roman Beltiukov, Walter Willinger, Ronaldo A. Ferreira, Arpit Gupta, and Lisandro Z. Granville. 2022. AI/ML for Network Security: The Emperor has no Clothes. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.).

[19] Ben Jacobsen, Sazzadur Rahaman, and Saumya Debray. 2021. Optimization to the Rescue: Evading Binary Code Stylometry with Adversarial Use of Code Optimizations. In *Proceedings of the 2021 Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks*. Association for Computing Machinery, 1–10. https://doi.org/10.1145/3465413.3488574

[20] C Robert Jacobsen and Morten Nielsen. 2013. Stylometry of paintings using hidden Markov modelling of contourlet transforms. *Signal Processing* 93, 3 (2013), 579–591.

[21] Patrick Juola. 2013. How a computer program helped reveal JK Rowling as author of A Cuckoo's Calling. *Scientific American* 20 (2013), 13.

[22] Vaibhavi Kalgutkar, Ratinder Kaur, Hugo Gonzalez, Natalia Stakhanova, and Alina Matyukhina. 2019. Code authorship attribution: Methods and challenges. *ACM Computing Surveys (CSUR)* 52, 1 (2019), 1–36.

[23] R Kelly Garrett. 2006. Protest in an information society: A review of literature on social movements and new ICTs. *Information, communication & society* 9, 02 (2006), 202–224.

[24] Gilles Louppe, Louis Wehenkel, Antonio Sutera, and Pierre Geurts. 2013. Understanding variable importances in forests of randomized trees. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) *(NIPS'13)*. Curran Associates Inc., Red Hook, NY, USA, 431–439.

[25] Scott Lundberg. 2024. SHAP Kernal Explainer. https://shap.readthedocs.io/en/latest/generated/shap.KernelExplainer.html

[26] Scott M. Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 4768–4777.

[27] Zaheed Mahmood, David Bowes, Tracy Hall, Peter C. R. Lane, and Jean Petric. 2018. Reproducibility and replicability of software defect prediction studies. *Inf. Softw. Technol.* 99 (2018), 148–163.

[28] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How Machine Learning Is Solving the Binary Function Similarity Problem. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 2099–2116.

[29] Xiaozhu Meng. 2016. Fine-grained binary code authorship identification. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 1097–1099.

[30] Xiaozhu Meng, Barton P Miller, and Kwang-Sung Jun. 2017. Identifying multiple authors in a binary program. In *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22*. Springer, 286–304.

[31] Jose G. Moreno-Torres, Troy Raeder, Rocío Alaíz-Rodríguez, Nitesh V. Chawla, and Francisco Herrera. 2012. A unifying view on dataset shift in classification. *Pattern Recognit.* 45, 1 (2012), 521–530.

[32] Frederick Mosteller and David L Wallace. 1963. Inference in an authorship problem: A comparative study of discrimination methods applied to the authorship of the disputed Federalist Papers. *J. Amer. Statist. Assoc.* 58, 302 (1963), 275–309.

[33] Hanchao Qi and Shannon Hughes. 2011. A new method for visual stylometry on impressionist paintings. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2036–2039.

[34] Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading authorship attribution of source code using adversarial learning. In *28th USENIX Security Symposium (USENIX Security 19)*. 479–496.

[35] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. Unleashing the hidden power of compiler optimization on binary code difference: an empirical study. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 142–157.

[36] Ross Ridge. 2014. Why do linked binaries contain the file names of used object files, how to remove them? `https://stackoverflow.com/questions/25457447/why-do-linked-binaries-contain-the-file-names-of-used-object-files-how-to-remov/25468506#25468506`.

[37] Margaret E Roberts. 2020. Resilience to online censorship. *Annual Review of Political Science* 23 (2020), 401–419.

[38] Nathan Rosenblum, Xiaojin Zhu, and Barton P Miller. 2011. Who wrote this code? identifying the authors of program binaries. In *Computer Security–ESORICS 2011: 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings 16*. Springer, 172–189.

[39] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. 2012. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 65–79.

[40] Saloni Alias Puja Sarnot, Sanjana Rinke, Rayomand Raimalwalla, Raviraj Joshi, Rahul Khengare, and Purvi Goel. 2019. Snapcode–a snapshot based approach to code stylometry. In *2019 International Conference on Information Technology (ICIT)*. IEEE, 337–341.

[41] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. 2002. Disassembly of executable code revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering*. IEEE, 45–54.

[42] Robin Sommer and Vern Paxson. 2010. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. In *31st IEEE Symposium on Security and Privacy, SP 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 305–316.

[43] Emily Stacey. 2015. The Pamphlet Meets API: An Overview of Social Movements in the Age of Digital Media. *Promoting Social Change and Democracy through Information Technology* (2015), 1–25.

[44] Ningfei Wang, Shouling Ji, and Ting Wang. 2018. Integration of static and dynamic code stylometry analysis for programmer de-anonymization. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*. 74–84.

[45] Daniel Watson. 2019. *Source Code Stylometry and Authorship Attribution for Open Source*. Master's thesis. University of Waterloo.

[46] Cynthia Whissell. 1996. Traditional and emotional stylometric analysis of the songs of Beatles Paul McCartney and John Lennon. *Computers and the Humanities* 30 (1996), 257–265.