

EVADING STYLISTIC ANALYSIS OF BINARY PROGRAMS USING  
ADVERSARIAL LEARNING

By

BENJAMIN RICHARD JACOBSEN

---

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree  
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

M A Y 2 0 2 1

Approved by:

---

Dr. Saumya Debray  
Department of Computer Science

# Evading Stylistic Analysis of Binary Programs Using Adversarial Learning

Benjamin Jacobsen

May 3, 2021

**Abstract** — Recent work suggests that it may be possible to determine the author of a binary program simply by analyzing stylistic features preserved within it. As this poses a threat to the privacy of programmers who wish to distribute their work anonymously, we consider steps that can be taken to mislead such analysis. We begin by exploring the effect of non-standard compiler optimizations on the features used for stylistic analysis. Building on these findings, we propose a black-box attack on a state-of-the-art classifier using compiler optimizations. Finally, we discuss our results, as well as implications for the field of binary stylometry.

## 1 Introduction

Think, for a moment, of your favorite writer. How would you describe their style? Do they write in short sentences, direct and unambiguous? Or is their style more lyrical and poetic? Is their vocabulary functional, or arcane? Chances are, if someone showed you one of their works which you were unfamiliar with, you would be able to spot the similarities almost immediately.

Every author has a unique style, and like a fingerprint, this style can be used to identify their works. Mosteller and Wallace used authorial style to investigate the authorship of the Federalist Papers [18]; Eder used it to discriminate between the work of different scribes in a translation of the Bible into Polish [9]; and in 2013, stylistic analysis was used to confirm that the crime novel *A Cuckoo's Calling* was in fact written by J. K. Rowling under a pseudonym [14]. These are only a few of the many and varied applications of the study of authorial style, or stylometry.

Of course, anyone who has dealt much with computer code understands that different programmers can have very distinctive styles as well, much the same as authors of prose. And so this raises a very

natural question — could the same sorts of techniques be used to identify the authors of computer programs?

This question has been investigated in a myriad of ways since the early 70s, and in general the answer seems to be *yes*, with several important caveats [15]. Unlike prose, code is valued for what it does, more than how it reads, and this simple fact introduces many complications. It is not uncommon for code to be written by multiple authors, or to incorporate snippets written by many programmers, perhaps copied from sites like StackOverflow. In addition, many tools exist which will automatically standardize features like indentation, which might otherwise carry stylistic information. At the extreme end, code might be obfuscated or compiled before being distributed in a form which is vastly different from the original, raw source code. Many researchers have proposed strategies for dealing with each of these issues, but for the most part they remain challenging open problems for the field [15]. Nonetheless, in controlled settings, researchers working with raw source code have attained accuracy of more than 95% when discriminating between over 200 authors [7].

With this in mind, it is worth considering the possible consequences of this technology. On the one hand, code stylometry could be a useful tool in resolving copyright disputes, accusations of plagiarism, and similar issues. But simultaneously, it poses an undeniable privacy threat. There are many legitimate reasons for a programmer to wish to distribute their code anonymously. For example, programmers who write tools to circumvent the surveillance of repressive governments might face serious personal risk if they were to be exposed.

More generally, writing code is a powerful means of participating in or contributing to a community. An activist might write a tool to track legislation, or organize protests. Someone involved in a support group might develop an app to help others struggling with their same conditions. Someone who identifies as part of a sexual minority might design a social net-

work targeted at others with the same orientation. In all three cases, the programmers involved might have good reason to fear discrimination or harassment should their involvement in the project become public knowledge. In these contexts, freedom of association is rather hollow without corresponding rights to control who knows about that association. And so we should take this sort of privacy threat seriously, and investigate the options available to programmers who wish to preserve their anonymity.

This thesis will deal with a specific instance of this problem, involving the distribution of binary programs. Recent work suggests that stylistic features can survive compilation [6], allowing the author to be identified. However, most prior work has dealt entirely with unoptimized binaries, or binaries compiled with one of a small number of standard bundles of optimization. It therefore remains an open question whether stylometric analysis is robust in the face of the full array of optimizations made available by modern compilers.

With this motivation, we investigate whether it is feasible for a privacy-conscious programmer, armed only with a standard compiler, to strategically choose optimizations in a way which obscures their tracks and misleads attribution. We first reproduce the work of Caliskan et al. [6], which represents a state-of-the-art open source binary stylometry system. We then propose a black-box attack on a stylometric model which makes use of global optimization techniques to search for combinations of optimizations which can obscure the stylistic traces left in the resulting binary. Finally, we evaluate the effectiveness of our attack on models trained using Caliskan et al.’s method.

## 2 Background

### 2.1 Code Stylometry

Code stylometry is the process of using the stylistic features of a program to determine who wrote it. This is typically modeled as a supervised machine learning problem.

A researcher begins by assembling a training dataset consisting of many programs, all of which have known authorship. From each program, the researcher extracts features which succinctly describe it. These might be *lexical* features (such as indentation style or function names), *syntactic* features (such as a preference for certain data structures or tendency to write longer or shorter functions), or *semantic* features (such as the actual algorithms implemented, or the overall flow of control) [15]. In whatever combina-

tion, these features are paired with a label indicating the author of the program and fed to some variety of machine learning algorithm, which learns how to discriminate between the different authors in the data. The final result is a model, which can be used to predict the author of new programs.

### 2.2 Adversarial Machine Learning

Machine learning techniques generally operate by studying a large amount of data drawn from some statistical distribution, and learning to recognize patterns within that data which help it solve some task at hand (say, distinguishing spam from legitimate email). Once these patterns have been learned, they can be used to classify new inputs drawn from the same distribution.

Attacking these models generally boils down to violating the assumption that the inputs to the model are drawn from the same distribution as the training data. This might involve tampering with the training data (data poisoning), or carefully crafting inputs that fool the classifier (called adversarial examples). In both cases, the goal is to cause the classifier to behave incorrectly in certain cases, for example by allowing spam into a recipient’s inbox. Despite the success of machine learning in other domains, many common machine learning techniques have been shown to be extremely vulnerable to these sorts of attacks, and devising methods for more robust learning remains a major open problem [4].

In categorizing different types of attacks, one important variable is the extent of the attacker’s knowledge of the system they are targeting. At one extreme, the attacker is assumed to know everything — the data used, the features extracted, the type of classifier, and so on. This model, which is generally called *white-box*, represents a worst-case scenario for the defender. At the other extreme, the attacker knows only high-level information about what a classifier is supposed to do, and the only way they can learn about its inner workings is by feeding it inputs and seeing what it does. This model is appropriately called *black-box*, and is the model that we adopt for our own attack.

### 2.3 Bayesian Optimization

Bayesian optimization is a technique for black box global optimization. That is, if we are allowed to query a function repeatedly, but otherwise have no access to its inner workings, Bayesian optimization can be used to search effectively for the input which maximizes (or minimizes) the output. In our case, we

use it to find the maximum of the function “Given some set of compiler optimizations, return the accuracy of the target model when classifying a binary compiled with those optimizations.”

For a more technical description of Bayesian optimization, the reader is referred to Peter Frazier’s excellent tutorial [13]. This is the high level concept:

At each stage of the algorithm, we have access to a record of all of our previous queries. Using these inputs and outputs, we use statistical inference to create a model of what we think our objective function looks like. The core assumption here is that points that are close to those we’ve already queried are likely to have similar outputs, and we can be more certain about the output we would get at a point the closer it is to points that we’ve already checked.

From here, we need to decide the next point to query. In order to leverage our earlier queries, we want to focus on points near the highest-value points we’ve found. But simultaneously, because we want to find the global optimum, we also want to look at points where we’re very uncertain what we might get. To balance these competing goals, we define an acquisition function over our domain which gives a certain weight to each priority, and choose our next point by finding the maximum of this acquisition function. The acquisition function is chosen so that it is easy to find the true, global maximum in a short amount of time.

Finally, once we find the point that maximizes our acquisition function, we query our objective function there, record the result, and update our data. This process can be repeated as often as desired, or until some computational budget is exceeded.

Bayesian optimization excels when querying our objective function is very expensive. In this situation, it’s worthwhile to take the time to construct a surrogate model and optimize an acquisition function over it. Choosing our next point in such a careful manner lets us cut down on the total number of queries we need to make, saving time and resources.

The greatest difficulty with Bayesian optimization is in scaling to higher dimensions [13]. Intuitively, in high-dimensional spaces, everything is far away from everything else (a phenomenon poetically called the Curse of Dimensionality). So, even after many queries, we still might be very uncertain of the value of our objective function for most inputs.

### 2.3.1 REMBO

REMBO is an extension of Bayesian optimization proposed by Wang et al. [29]. In many real-world situations, a problem can appear to be very high-

dimensional, when in fact its *intrinsic* dimensionality is quite small. That is, it may be the case that only a few of the dimensions affect the objective function significantly. For example, the hyperparameters of neural networks have been found to have this property [3].

The key insight of Wang et al. is that it is possible to take advantage of this structure when optimizing such a function, even when we do not know exactly which of the hyperparameters are actually relevant. Thus, as long as we know that the intrinsic dimensionality of a function is small (say, 10), we can find its optimum almost as easily as we could a normal 10-dimensional function, even if we don’t know which 10 dimensions matter.

An example is useful to illustrate this surprising fact. Suppose we want to find the optimum of a function with two parameters,  $f(x, y)$ . We suspect that only one of these parameters matter, but we don’t know which one. One solution is to simply look for optimums on the line  $x = y$ . This reduces the space we have to search from 2 dimensions to 1, and our space is still guaranteed to contain the optimal value regardless of whether  $x$  or  $y$  is the important dimension.

What if all the variation in our objective function lies in a 1-dimensional subspace that isn’t aligned to either  $x$  or  $y$ ? Then we can simply choose a line to optimize along at random, and we will only fail to find our optimum if the line we choose just happens to be exactly perpendicular to the true 1-dimensional subspace, which happens with probability 0.

This same reasoning can be extended to higher dimensions. The upshot is that we can use Bayesian optimization for very high dimensional functions as long as there is compelling reason to believe that the intrinsic dimensionality is small. In section 3, we study the impact of different compiler optimizations on binary stylometry, and conclude that it appears to fit this pattern.

## 2.4 Compiler Provenance

Compiler provenance refers to the problem of determining the exact compiler and optimizations used to create a particular binary. This involves identifying the family of compiler (e.g. GCC vs. Clang), the version used (e.g. GCC 3.4.x vs. GCC 4.4.x), and the optimizations employed (e.g. O0 vs. O2). Recent work in this area has produced tools which are capable of determining the difference between optimized and unoptimized binaries with high accuracy [26] [24]. However, compilers like GCC are capable of performing on the order of 200 independent optimizations,

and no work has come remotely close to being able to distinguish between the corresponding  $\sim 2^{200}$  possible combinations.

This poses a difficulty for binary stylometry. Recall that classifiers generally must be trained on data drawn from the same distribution that they will eventually be employed on. Thus, to avoid accidentally classifying particular compilers or optimizations instead of authors, it is important to use a classifier trained on programs compiled under the same conditions as the program being studied. This may be feasible when that program has been compiled with very standard options, such as O0 and O2, but is currently impossible to guarantee in general.

## 2.5 Google Code Jam Dataset

In all of our experiments, we used data from the Google Code Jam (GCJ), which is a popular international coding competition hosted by Google. This dataset is standard in much of the literature around program stylometry (CITATIONS?) because it is one of the few large corpuses of programs that:

1. Are known to be written by specific, single authors
2. Don't contain 3rd party or copy-pasted code
3. Are attempting to perform the same task

These features make it in some sense ideal for analyzing programming style. Specifically, our dataset consists of 200 authors who participated in the 2017 GCJ. Then, for each author, our dataset contains their submissions to 8 of the problems from the coding competition. This data was originally collected by Quiring et al. [23], who made it publicly available. All submissions were confirmed to be correct.

## 3 Preliminary Exploration

### 3.1 Most optimizations have little effect

Our largest concern at the outset was that programs which are too small and simple might simply lack the features that are required for compiler optimization. This would render any attempt to manipulate stylistic features of such programs doomed from the start. To test this possibility, we performed the following simple experiment:

First, we randomly sampled 20 programs from the Google Code Jam dataset. Then, for every optimization flag provided by GCC 7.5.0, we compiled each

one of those programs. Next, we decompiled each of the resulting binaries using the Ghidra decompiler. We performed the same process using no optimizations at all. Then, after filtering out boilerplate code inserted by the decompiler, we compared the code under each given optimization with the baseline of no optimization at all using the `diff` tool and recorded the number of lines which changed.

The result was that, while a small handful of optimizations had a very substantial effect on the resulting decompiled code, the vast majority had no effect whatsoever. Of the roughly 200 optimizations tested, only 32 changed a single line of code in any instance, and only 13 changed more than 10 lines of code on average.

This analysis should be taken with a grain of salt, as it might be both under and over-inclusive. On the one hand, because we only tested optimizations in isolation, it is quite likely that we overlooked optimizations which can operate only when paired with other optimizations. Simultaneously, some optimizations which change many lines of code might not adversely influence stylometry. Nonetheless, based on these results, it is at least plausible that A) optimization can have a significant effect even on small programs, and B) only a small portion of the optimizations provided by GCC have a large effect on the resulting binary.

### 3.2 Optimization perturbs informative features

While it is clear that different choices of optimizations can lead to different binaries being produced, it is not immediately obvious to what extent these differences might interfere with stylometry. In particular, it is conceivable that the features which carry the most stylistic information might be particularly resilient to being manipulated through optimization.

To investigate this possibility, we decided to focus on the distribution of opcode frequencies, as this was reported by Caliskan et al. [6] as being one of the most informative sources for stylistic features. We compiled the 2017 Google Code Jam dataset with eleven different optimization flags, chosen to cover the range of common optimizations while also including architecture-specific optimizations. We then disassembled each of the resulting binaries using `objdump` and extracted the frequency of each opcode 2-gram in the disassembly. Finally, we used `scikit-learn` [21] to extract the 50 2-grams most useful for stylometry, using information gain. We then studied the extent to which these frequencies were perturbed by optimization, using two different methods for measuring that perturbation.

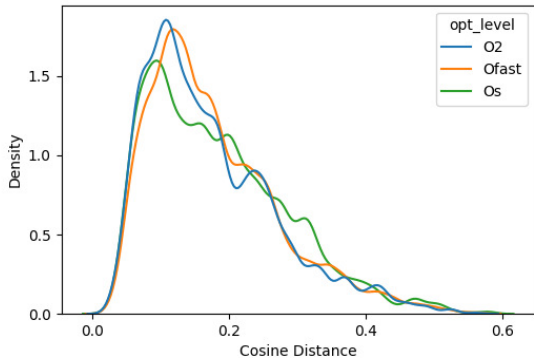


Figure 1: Distribution of the cosine distance between optimized and unoptimized versions of the same program, considering only the 50 most informative opcode ngrams.

### 3.2.1 Perturbation as Cosine Distance

Cosine distance is a measure of the similarity of two vectors, defined to be  $1 - \cos \theta$ , where  $\theta$  is the angle between the two vectors. When used on vectors whose components are all non-negative (such as vectors of frequencies), this value always lies between 0 and 1. A distance of 0 indicates that the two vectors have the same orientation, and differ only in length, if at all. Conversely, a distance of 1 indicates that the two vectors are orthogonal. In the context of frequency vectors, this would imply that the any feature which appears in one observation is absent in the other, and visa versa.

Because cosine distance measures angle and not magnitude, it is particularly useful for comparing vectors that can vary considerably in size. We might, for example, expect two programs written by the same author to have similar distributions of opcode ngrams. However, if one program is larger than the other, then the raw numbers may be very different. The cosine distance between these two programs would be quite small, whereas it might be quite large for a metric like Euclidean distance.

Using cosine distance, we can measure the size of the perturbation caused by compiling a program with a certain level of optimization. Using kernel density estimation, we then estimate the distribution of distances between binaries optimized with a certain flag and their unoptimized counterparts.

This distribution turns out to be very right-skewed. The median cosine distance is roughly 0.16 at all three levels of optimization tested, but a meaningful proportion of programs showed extreme variation. Almost all programs were perturbed to some degree.

	O2 v. Ofast	O2 v. Os	Ofast v. OS
$L_{0.025}$	0.075	0.213	0.291
$L_{0.975}$	0.238	0.536	0.6

Table 1: Empirical 95% confidence intervals for the proportion of authors who change clusters when compiled at one level of optimization over another.

### 3.2.2 Perturbation as Changing Clusters

Our results from the prior section suggest that, in many cases, optimization can meaningfully perturb the frequency of opcode ngrams. However, we might imagine that programmers with different styles could be quite distinct to begin with, and so programs written by one author could remain identifiable even if they move a considerable distance.

To investigate this possibility, we reduced the dimensionality of our data from 50 down to 2 using principal component analysis, sampled 10 authors at random, and used k-means clustering with 10 centroids to separate programs written by those authors into clusters. This clustering was done using only data from a single level of optimization. Then, for each other level of optimization, we observed the proportion of times that a program compiled at that level fell into a different cluster than it did at our baseline level. By repeating this process 1000 times with independent samples of authors, we derived empirical 95% confidence bounds for the proportion of programs which will change clusters.

We found that the difference between unoptimized and optimized versions of the same program was consistently quite large, with between 50% and 75% of programs changing clusters when any optimization is applied. Large differences also exist between binaries compiled at different levels of optimization, however, as shown in Table 1.

Full data from both of these experiments are available on github<sup>1</sup>. On the basis of these results, we believe that there is compelling reason to suspect that compiler optimizations could substantially interfere with binary stylometry. These suspicions are tested more rigorously in section 4.

## 3.3 Replication of Caliskan et. al

We also worked to replicate the findings of Caliskan et al. [6]. Using their publicly provided code, we were able to build a stylometric classifier with 90% accuracy, using a dataset of 20 authors and 8 programs per author.

<sup>1</sup><https://github.com/skdebray/Stylometry>

This number is meaningfully lower than the 99% accuracy reported by Caliskan et al. under the same conditions, but similar to the figure reported by Meng et al. in their 2018 replication of Caliskan’s work [16]. Both our work and Meng’s used a 64-bit platform, while Caliskan et al. studied 32-bit platforms, which may account for some of the difference.

## 4 Evaluation

### 4.1 Attack Methodology

We assume only black-box access to our target model. That is, we are allowed to compile our program(s) with whatever flags we choose, submit them to the classifier, and observe which author they are attributed to. We do not have access to the set of features considered by the model, or to the confidence of its attributions.

Moreover, we assume that the defender is able to perform a limited degree of compiler provenance. Because current compiler provenance technology is only able to distinguish relatively coarse levels of optimization, as discussed in the background section, we permit the defender to use four classifiers, trained on datasets compiled with -O0, -O1, -O2, and -O3 respectively.

To avoid the added complexity of actually performing compiler provenance on each binary we create as part of our attack, we simply feed our program(s) to each classifier in turn, and use the highest accuracy of any classifier in the ensemble to quantify the effectiveness of the attack. In this way, we model a defender which is capable of identifying with perfect accuracy which of the four coarse levels of optimization best matches our input program.

The attacker is presumed to have access to a number of programs written by authors known to the classifier. This assumption represents the plausible scenario where the attacker is the author attempting to avoid attribution, and therefore has ready access to other code that they’ve written. This allows the attacker to receive more fine-grained information from the classifier based on the proportion of programs that are misclassified under a given set of optimizations.

Under these circumstances, the goal of the attacker is to repeatedly query the defender’s models, using the results to find combinations of optimizations which can mislead attribution. Because even a single query takes a non-trivial amount of time, the attacker would ideally like to find such a combination within a reasonably small number of queries.

### 4.2 Experimental Design

For a single attack, we randomly sample 10 authors from the Google Code Jam dataset. We then split this sample into a ‘training’ dataset, consisting of 7 programs per author, and a ‘testing’ dataset, with the remaining 1 program per author. To avoid complications from a single problem being particularly ill-suited for stylometry (for example, because it is very simple), we ensure that the ‘testing’ programs are chosen evenly between the 8 problems in the dataset.

We then compile the training dataset four times, using the -O0, -O1, -O2, and -O3 flags, and train a classifier on each instance of the dataset. The end result is four models, which form the ensemble we will attack.

Next, we define our objective function. The input to this function is a binary vector, indicating which of GCC’s many optimizations to use. To evaluate the function, we compile the testing dataset with the given set of optimizations, and submit all 10 binaries to our ensemble of classifiers. The proportion of programs misclassified by the *most* accurate classifier is the output of our function.

To actually carry out our attack, we maximize this objective function using Bayesian Optimization, and specifically using the REMBO platform. Recall that REMBO is able to substantially improve the efficiency of Bayesian Optimization on high-dimensional functions (such as our objective function) when the intrinsic dimensionality is low. On the basis of our exploratory findings described in Section 3.1, we guess that the intrinsic dimensionality of our problem is 20.

Finally, we assign a computational budget of 100 total iterations. The success of the attack is the maximum value of the objective function that it managed to locate.

### 4.3 Results

Owing to the large computational cost of feature extraction, running a single attack takes on the order of 10 days. As a consequence, we have only run one attack to completion, and at the time of writing have another underway. As such, our results at this point are provisional.

In both cases, the classifier models attained an accuracy of roughly 85% on the training dataset, with 7 instances per author. When applied to the testing dataset compiled with an initial, random set of optimizations, this accuracy dropped immediately to 50%, and subsequently fell to 40% after around two-dozen iterations, which was not later improved on. In our second, ongoing attack, the initial accuracy was only 20%, where it remains after 31 iterations.

This result is far from statistically definitive by virtue of the small sample size. However, the large drop in accuracy relative to the training dataset does suggest that the use of optimizations beyond standard levels like O2 can present difficulties for binary stylometry.

## 5 Discussion

Prior work on binary stylometry has assumed full knowledge of the tool chain used to produce the target binary, including the exact optimizations used [6]. However, as discussed in section 2.4, this is in general not possible to guarantee. Our results suggest provisionally that violating this assumption by using non-standard optimizations can substantially reduce classification accuracy. In fact, even without black-box access to a stylometric classifier, simply choosing sets of optimizations uniformly at random may be an effective and easily-implemented strategy for programmers to obscure their stylistic fingerprint.

Several interesting questions arose during the course of this work which could not be answered because of constraints on time and resources. In particular, because of the large computational cost associated with even a single attack, we were only able to perform a single full attack in this work. To understand the feasibility of adversarial attacks on stylometric tools more deeply, it will be necessary to replicate our experiment under many different conditions, including datasets of different size and from different sources.

## 6 Related Work

### 6.1 Code Stylometry

Formal research into code stylometry began in the 1970s, where it was primarily focused on the problem of plagiarism detection [20] [8]. This research focused on manually searching for comprehensible measures of code similarity, which were then empirically validated on datasets of student code. Early emphasis was placed on lexical features, such as counting the number of operands or lines of code. Later, researchers shifted towards syntactic features, such as a preference for certain data structures, which are more resilient to sophisticated plagiarism methods [10] [30] [27].

Study of authorship attribution — that is, research premised on the idea that authors have unique and identifiable fingerprints — began in the late 80s [19], and was strongly influenced by the work of Spafford

and Weeber[28] in 1993. Spafford and Weeber were among the first to consider authorship attribution in an adversarial context, investigating whether it could be used to identify the authors of malware.

In these early decades, focus was placed on easily interpreted features, chosen manually by human researchers. In 2006, Frantezkou et al. revolutionized the field by proposing the use of byte-level ngrams [11][12], which they showed to be highly effective for stylometry. Subsequent research has largely followed in their footsteps, automatically extracting relevant features from their data rather than defining them in advance. In addition to ngrams, features extracted from the abstract syntax tree (AST) of a program have also become standard [2] [7] [22].

The current state of the art in source-level stylometry is represented by Caliskan et al., who use a wide range of automatically-extracted features to classify 1600 authors with 94% accuracy [7]. More recently, Abuhamad et al. report 92% accuracy in classifying 8903 authors, using primarily lexical features and recurrent neural networks [1].

### 6.2 Binary Stylometry

The application of stylometry to binary programs is more recent. Rosenblum et al. were among the first to consider the problem in detail, and found that programmer style appeared to meaningfully survive the compilation process [25]. Caliskan et al., whose work is the subject of the attack described in this thesis, improved on Rosenblum’s results through the use of a random forest classifier [6]. Meng et al. consider the problem in the context of programs written by multiple authors, and propose fine-grained stylometric analysis at the level of individual basic blocks [17].

### 6.3 Evasion of Stylometry

Despite growing interest in the application of stylometry to adversarial scenarios, relatively little work has considered the application of adversarial machine learning to evade stylistic classification. Brennan et al. considered manual attacks on literary stylometry, but did not find effective automatable methods for evasion [5]. In 2018, Meng et al. were among the first to study adversarial attacks on binary stylometry by altering a pre-existing binary, and achieved a high success rate in untargeted attacks [16]. Quiring et al. presented the first automated attack on source-level stylometry in 2019, using the Monte-Carlo Tree Search method alongside several hand-crafted code transformations to create plausible-looking adversarial examples [23].

## References

- [1] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. Large-scale and language-oblivious code authorship identification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 101–114, 2018.
- [2] Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. Source code authorship attribution using long short-term memory based networks. In *European Symposium on Research in Computer Security*, pages 65–82. Springer, 2017.
- [3] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.
- [4] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.
- [5] Michael Brennan, Sadia Afroz, and Rachel Greenstadt. Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity. *ACM Transactions on Information and System Security (TISSEC)*, 15(3):1–22, 2012.
- [6] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. *arXiv preprint arXiv:1512.08546*, 2015.
- [7] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 255–270, 2015.
- [8] John L Donaldson, Ann-Marie Lancaster, and Paula H Sposato. A plagiarism detection system. In *Proceedings of the twelfth SIGCSE technical symposium on Computer science education*, pages 21–25, 1981.
- [9] Maciej Eder. Rolling stylometry. *Digital Scholarship in the Humanities*, 31(3):457–469, 2016.
- [10] Jinan AW Faidhi and Stuart K Robinson. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education*, 11(1):11–19, 1987.
- [11] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. Effective identification of source code authors using byte-level information. In *Proceedings of the 28th international conference on Software engineering*, pages 893–896, 2006.
- [12] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. Source code author identification based on n-gram author profiles. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pages 508–515. Springer, 2006.
- [13] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.
- [14] Patrick Juola. How a computer program helped reveal jk rowling as author of a cuckoo’s calling. *Scientific American*, 20:13, 2013.
- [15] Vaibhavi Kalgutkar, Ratinder Kaur, Hugo Gonzalez, Natalia Stakhanova, and Alina Matyukhina. Code authorship attribution: Methods and challenges. *ACM Computing Surveys (CSUR)*, 52(1):1–36, 2019.
- [16] Xiaozhu Meng, Barton P Miller, and Somesh Jha. Adversarial binaries for authorship identification. *arXiv preprint arXiv:1809.08316*, 2018.
- [17] Xiaozhu Meng, Barton P Miller, and Kwang-Sung Jun. Identifying multiple authors in a binary program. In *European Symposium on Research in Computer Security*, pages 286–304. Springer, 2017.
- [18] Frederick Mosteller and David L Wallace. Inference in an authorship problem: A comparative study of discrimination methods applied to the authorship of the disputed federalist papers. *Journal of the American Statistical Association*, 58(302):275–309, 1963.
- [19] Paul W Oman and Curtis R Cook. Programming style authorship analysis. In *Proceedings of the 17th conference on ACM Annual Computer Science Conference*, pages 320–326, 1989.
- [20] Karl J Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM Sigcse Bulletin*, 8(4):30–41, 1976.
- [21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher,

- M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [22] Brian N Pellin. Using classification techniques to determine source code authorship. *White Paper: Department of Computer Science, University of Wisconsin*, 2000.
- [23] Erwin Quiring, Alwin Maier, and Konrad Rieck. Misleading authorship attribution of source code using adversarial learning. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 479–496, 2019.
- [24] Ashkan Rahimian, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation*, 14:S146–S155, 2015.
- [25] Nathan Rosenblum, Xiaojin Zhu, and Barton P Miller. Who wrote this code? identifying the authors of program binaries. In *European Symposium on Research in Computer Security*, pages 172–189. Springer, 2011.
- [26] Nathan E Rosenblum, Barton P Miller, and Xiaojin Zhu. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 21–28, 2010.
- [27] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, 2003.
- [28] Eugene H Spafford and Stephen A Weeber. Software forensics: Can we track code to its authors? *Computers & Security*, 12(6):585–595, 1993.
- [29] Ziyu Wang, Masrour Zoghi, Frank Hutter, David Matheson, Nando De Freitas, et al. Bayesian optimization in high dimensions via random embeddings. In *IJCAI*, pages 1778–1784, 2013.
- [30] Geoff Whale. Identification of program similarity in large populations. *The Computer Journal*, 33(2):140–146, 1990.