Optimization to the Rescue: Evading Binary Code Stylometry with Adversarial Use of Code Optimizations

Ben Jacobsen
bjacobsen@email.arizona.edu
University of Arizona
Tucson, Arizona, USA

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

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

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 compiler optimizations on the features used for stylistic analysis. Building on these findings, we propose a gray-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.

CCS CONCEPTS

• **Security and privacy** → **Privacy protections**; • **General and reference** → *Empirical studies*; *Experimentation*; • **Software and its engineering** → *Compilers*; • **Computing methodologies** → **Machine learning**; • **Social and professional topics** → *Surveillance.*

KEYWORDS

Privacy; Adversarial Machine Learning; Stylometry; Bayesian Optimization

ACM Reference Format:
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 (Checkmate '21), November 19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3465413.3488574

1 INTRODUCTION

The analysis of individual stylistic characteristics has been used to infer the authorship of natural-language texts [18, 25, 33]. The idea has also been applied to software, focusing on the analysis of stylistic clues in software to identify its possible authors [1, 4, 9, 10, 20, 21, 32, 36, 41, 46, 50]. Software, unlike natural-language prose, has characteristics that can make stylistic analysis challenging: for example, it can be written by multiple authors or incorporate code snippets obtained from sites like StackOverflow. There are tools that automatically standardize features, like indentation, which might otherwise carry stylistic information. 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 [26]. Nonetheless, in controlled settings, researchers working with raw source code have attained accuracy of more than 95% when discriminating between over 200 authors [10]. Recent work suggests that stylistic features may also survive compilation, allowing the author to be identified from binaries [9]. However, most prior work on this topic has typically considered only a few "standard" optimization flags. 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.

Note that while it is obvious that in, general, compiler optimizations can profoundly alter many of the low-level characteristics of compiled code, it is not *a priori* obvious that such optimizations can sufficiently erase *all* of the stylistic features characteristic of an individual programmer for a particular program (see Section 3.4). This makes it hypothetically possible that enough stylistic clues might survive the optimization process to allow that program's author to be identified.

While code stylometry can be a useful tool in some circumstances, e.g., for resolving copyright disputes or identifying malware authors, it also 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 software tools to circumvent the surveillance or censorship of repressive governments, help activists track legislation and organize protests, or help members of oppressed minorities, might all have reason to fear discrimination or harassment should their involvement in such projects become public knowledge.

This paper aims to address this privacy challenge, focusing on compiled binaries, which are commonly used to distribute software. We reject the use of code obfuscation tools, since these can make maintenance problematic—and, more importantly from the perspective of a privacy-conscious developer, can potentially embed identifying markers into the obfuscated code in ways that may not be easy to detect (e.g., see [14]). Instead, we focus on leveraging conventional compiler optimizations to erase stylistic clues in compiled binaries. Importantly, our approach is usable by ordinary privacy-conscious programmers who need not be familiar with sophisticated machine learning concepts or tools based on such concepts [42]. Our approach is based on Bayesian optimization, but it is used only to recommend optimization flags to the programmer. The programmer can then research the recommended flags to understand what the optimizations do, apply them and examine the resulting binaries to determine their effect, and possibly adapt

Checkmate '21, November 19, 2021, Virtual Event, Republic of Korea.

ACM ISBN 978-1-4503-8552-7/21/11.
https://doi.org/10.1145/3465413.3488574

them in ways she prefers. This has two important advantages: (1) it is easily integrated into conventional software development practices; and (2) it provides transparency and does not demand any additional trust from the programmer on any "magic" software (e.g., obfuscators or anti-stylometry tools).

This paper makes the folowing contributions:

(1) We demonstrate experimentally that carefully chosen compiler optimizations can significantly alter the stylometric characteristics of compiled binaries and impact the accuracy of binary code stylometry. This indicates limits to the scope of earlier work on binary-level stylometry when applied to optimized binaries [9].
(2) We propose a method that is usable by ordinary programmers, using ordinary software development practices, to systematically evade binary-level stylometry.
(3) Experimental evaluation, based on a state-of-the-art open source binary stylometry system [9], shows that, our method can cause a significant drop (%) in stylometric accuracy.

2 BACKGROUND

2.1 Code Stylometry

Code stylometry is the process of using the stylistic features of a program to determine who wrote it [36, 41, 50]. This is typically modeled as a supervised machine learning problem [1, 4, 9, 10, 20, 21, 32, 46].

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) [26]. In whatever combination, 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 Compiler Optimization

Compiler optimizations aim to improve binary-level code metrics while preserving observable behavior. The code metrics most commonly used are execution speed and code size, though researchers have also considered energy usage [27, 38] and (in smart contracts) monetary cost [3, 13]. Not surprisingly, optimizations focus on program constructs that most impact the metric under consideration. Thus, optimizations aimed at improving execution speed typically focus on loops (e.g., loop unrolling, loop fusion, code motion out of loops, loop vectorization), memory accesses (register allocation), removal of redundant or unnecessary code (constant folding, dead code elimination, common subexpression elimination), etc. [2]. Optimizations aimed at improving code size focus on reducing code replication, e.g., via procedural abstraction [16].

2.3 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 [35]), or carefully crafting inputs that fool the classifier (called adversarial examples[12, 52]). 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 [5, 7, 53, 54].

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* [51], 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* [15, 39]. In this paper, we adopt a gray-box approach for our own attack model, which we explain in Section 5.

2.4 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 [22]. 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 have 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 have 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 have found. But simultaneously, because we want to find the global optimum, we also want to look at points where we are 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 [22]. 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.

2.4.1 REMBO. REMBO is an extention of Bayesian optimization proposed by Wang *et al.* [55]. 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 [6].

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 hyperparemeters 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 do not 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 do not 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.

3 DESIGN INTUITIONS

Before diving into the details of our work, in this section we present our intuitions behind leveraging compiler optimizations to build a framework to evade binary code stylometry. Specifically, we discuss how compiler optimizations affect binary-level code characteristics and how this can impact binary-level code stylometry. This can help us to provide a context to generalizations about binary-level stylometry of optimized code that are sometimes encountered in the research literature. Figure 1 shows the effects of some common compiler optimizations successively applied to a small example C program, shown in Figure 1(a).[1] Function inlining pulls the body of the function `f()` into the caller function (Figure 1(b)). This causes the iteration count of the `for`-loop to become known, allowing it to be unrolled (Figure 1(c)). Finally, constant folding on the unrolled loop allows the conditionals to be optimized away and results in the code shown in Figure 1(d). In this example, a similar effect could have been obtained using interprocedural constant propagation [11] instead of function inlining. This example illustrates the following key observations.

3.1 Optimizations affect core features

The characteristics of an optimized program can be very different from both the source code it was obtained from as well as those of the same program compiled with a different set of optimizations. For example, function inlining can significantly change the structure and size of function bodies, as shown in Figure 1(b); loop unrolling can replace loops with longer loop-free instruction sequences, as shown in Figure 1(c); and constant folding can get rid of conditional branches, as shown in Figure 1(d).

It follows from this that source-level code features such as loop structure or function size, which may be useful for source-level stylometry, may not survive optimization unscathed. For example, the loop and conditional in the original source code shown in Figure 1(a) are completely eliminated in the optimized code shown in Figure 1(d). A corollary is that for stylometry purposes, source-level code features such as function size or loop structure, extracted from optimized binaries that have been decompiled using tools such as Hex-Rays or Ghidra, may not correspond meaningfully to the original source code.

3.2 Impact of code structure on optimization

The impact of any given optimizing transformation is dependent on the characteristics of the code being optimized. For example, loop unrolling will not have any effect on a program that does not have any unrollable loops, and function inlining will not significantly impact programs with few inlinable functions.

It follows from this that when we consider the efficacy of binary-level code stylometry on optimized code, it is important to take into account both the kinds of optimizations being applied and also the characteristics of the code those optimizations are being applied to. There are two corollaries: first, inferences about the efficacy

[1] For ease of understanding the results of various optimization steps are shown in the form of C source code, although in reality the compiler would use an intermediate representation, such as three-address code organized into a control flow graph, to which the optimizations would be applied [2].

```c
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

```c
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

```c
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

```c
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 1: An example of the effect of compiler optimizations

of binary code stylometry based on a particular set of optimizations on a particular set of programs may or may not generalize to a different set of optimizations applied to a different set of programs; and second, optimizations tailored to the characteristics of a particular program can have a significantly greater impact on the characteristics of the optimized code than a generic set of optimizations that may or may not be relevant to the code characteristics of that program [45].

3.3 Interactions between optimizations

Compiler optimizations may not be independent of each other. This is illustrated in Figure 1, where constant propagation to eliminate the `if`-statement in the loop was made possible due to the prior application of loop unrolling, which in turn was enabled due to the loop iteration count becoming known via function inlining. Ren *et al.* observe that optimizations may also sometimes influence each other negatively [45].

3.4 Optimization and stylometry

As noted in Section 3.1, compiler optimizations can profoundly alter the features in the optimized code. However, this does not, in itself, imply that optimization can necessarily render binary-level stylometry ineffective. There are two reasons for this:

(1) As discussed in Section 3.2, the impact of an optimization is dependent on the structure of the code it is applied to. Thus, even if an optimization is very effective in altering low-level code features *in general*, the characteristics of a particular program may render that optimization ineffective *for that program.*

(2) Stylometric analyses rely on statistical analyses of multiple stylistic features to attribute authorship. Even if we assume that compiler optimizations can erase many of the stylistic features characteristic of a particular programmer, it is by no means obvious that they will be able to erase *all* such characteristic stylistic features from a particular program. In other words, it is possible that, for that particular program, enough stylistic features may survive optimization to allow authorship attribution.

3.5 The difficulty of provenance analysis

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 [43, 47]. 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.

4 PRELIMINARY EXPLORATION

In this section, we present preliminary experiments that we performed to assess the feasibility of our basic method. Binary stylometry relies on extracting certain informative features from a given binary. Our guiding research question was "do different compiler optimizations meaningfully influence the distribution of these features?" An affirmative answer to this question is a necessary condition to use compiler optimizations for obscuring stylistic features and protecting programmer privacy.

4.1 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, e.g. [4, 9, 10] because it is one of the few large corpuses of programs that:

(1) Are known to be written by specific, single authors
(2) Do not 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.* [42], who made it publicly available. All submissions were confirmed to be correct.

4.2 Impact of optimization on core 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 ngram frequencies, as this was reported by Caliskan *et al.* [9] 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.[2] Finally, we used scikit-learn [40] to extract the 50 2-grams most useful for stylometry, using information gain. The five most informative 2-grams are provided in Table 1 for illustrative purposes. We then studied the extent to which these frequencies were perturbed by optimization, using the measure of cosine distance.

[2]The use of n-grams with $n = 2$ in this example is intended only to illustrate the impact of compiler optimizations on binary-level code features; our experiments with other values of n show qualitatively similar results. We note that 2-grams play an important role in the binary-level stylometric analysis of Caliskan *et al.* [9].

	Opcode 2-gram
1	`mov mov`
2	`mov call`
3	`endbr64 push`
4	`call mov`
5	`lea mov`

Table 1: The 5 opcode 2-grams which carry the most stylistic information

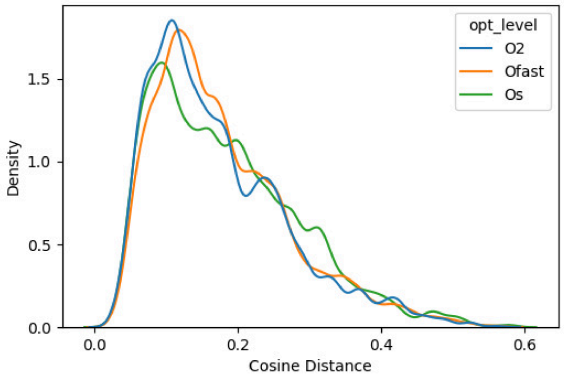


Figure 2: Distribution of the cosine distance between optimized and unoptimized versions of the same program, considering only the 50 most informative opcode ngrams.

Cosine distance is a measure of the similarity of two vectors, defined to be $1 - \cos\theta$, where θ 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 probability distribution of distances between binaries optimized with a certain flag and their unoptimized counterparts. Our results, showing the calculated distributions for three different levels of optimization, are presented in Figure 2.

From this figure, we can see that all three levels of optimization led to similar distributions of distances. These distributions turn out to be very right-skewed, with a median cosine distance of roughly 0.16 at all three levels of optimization tested. Concretely, the figure

tells us that almost all programs were perturbed to some degree, and typically the perturbation was moderate in size. However, some programs showed extreme variation. Full data from this experiment is available at `https://github.com/skdebray/Stylometry`. On the basis of these results, we believe that there is compelling reason to suspect that compiler optimizations could substantially interfere with binary stylometry.

5 EVADING CODE STYLOMETRY

In this section, we present our code optimization-based binary code stylometry evasion framework.

5.1 Threat Model

We assume that a programmer knows the crowd among which she wants to hide identity and also able to collect code samples from them. She can change the crowd when she wishes. We also assume that the programmer is skeptical about using any obfuscation or other black-box methods that are not transparent to her. Here, we refer the programmer as the attacker and the entity interested in binary code authorship attribution as the defender, since the programmer is potentially *attacking* the authorship attribution for evasion.

Current compiler provenance technology is only able to distinguish relatively coarse levels of optimization, as discussed in Section 3. Accordingly, we also assume that defender is able to perform a limited degree of compiler provenance, distinguishing between the optimization flags -O0, -O1, -O2, and -O3.

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. Figure 3 summaries our attack methodology.

5.2 Attack Methodology

To decouple our framework from any stylometric approach, we assume only limited access to our target stylometry model. We are allowed to compile our program with whatever flags we choose, submit it to the classifier, and observe which author it is attributed to and the confidence of that attribution. We do not have direct access to any structural information about the model, such as the set of features it uses.

Since the defender is able to perform compiler provenience for -O0, -O1, -O2, and -O3 optimization flags, we use an ensemble of four classifiers trained on datasets compiled with these four flags. To avoid the added complexity of actually performing compiler provenance on the binary, we simply feed our input program to each of these classifiers in turn, and use the lowest error rate 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.

Next, we define our objective function. The input to this function is a binary vector, indicating which of 192 of GCC's optimizations to use. Given the input program, our framework compiles it with the given set of optimizations, and submits it to our ensemble of classifiers. Our objective function is equal to 1 minus the confidence of the *most* confident correct attribution. If all of the classifiers in the ensemble misclassify the binary, then the attack is considered a success.

Formally, let $\vec{v} \in \mathbb{F}_2^n$ denote a binary vector corresponding to a certain set of optimizations, and let C denote an ensemble of classifiers. We model each classifier as a function $c_i \in C : \mathbb{F}_2^n \rightarrow [0, 1]$. Let a_i denote the confidence with which the classifier c_i correctly attributes a binary compiled with the given set of optimizations, or 0 if the binary is misclassified. Then $c_i(\vec{v}) := 1 - a_i$. Our objective is to find $\vec{v}$ such that:

$$\underset{\vec{v}}{\text{argmax}} f(\vec{v})$$

where

$$f(\vec{v}) = \ min(\{c(\vec{v})\}) \ \forall c \in C$$

For example, suppose we compile a program with a certain set of optimizations and feed it to our ensemble of classifiers. The classifiers trained on -O0 and -O1 both misclassify the binary, while the classifier trained on -O2 correctly attributes it with confidence 0.3, and the classifier trained on -O3 correctly attributes it with confidence 0.4. In this case, our objective function is 1 minus the largest of these values, or 0.6.

To actually carry out our attack, our framework maximizes this objective function using Bayesian Optimization, and specifically using the REMBO platform [55]. At each step of the process, REMBO outputs a binary vector indicating a set of optimizations to use, and is given the computed objective function in return, which helps determine the next set of optimizations to try. 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. We conservatively estimate that the intrinsic dimensionality of our problem is 20, as this is at the upper end of the range where Bayesian Optimization is believed to be effective [22].

6 EVALUATION

Our experimental evaluation seeks to evaluate the effectiveness of our authorship attribution evasion technique. Specifically, Our experimental evaluation answers the following research questions:

- Can our compiler optimization-based evasion method systematically generate adversarial examples?
- What is the impact of the number of iterations in Bayesian optimization on finding adversarial examples?

Next, we discuss our experimental setup, design and finally our evaluation results.

6.1 Experimental Setup

We used the Google code jam dataset that we used for our preliminary exploration (Section 4.1). We ran our experiment on a server with 32 cores (@ 3.30 Ghz) and 1 TB of RAM, running Ubuntu 20.04 and GCC 7.5.0.

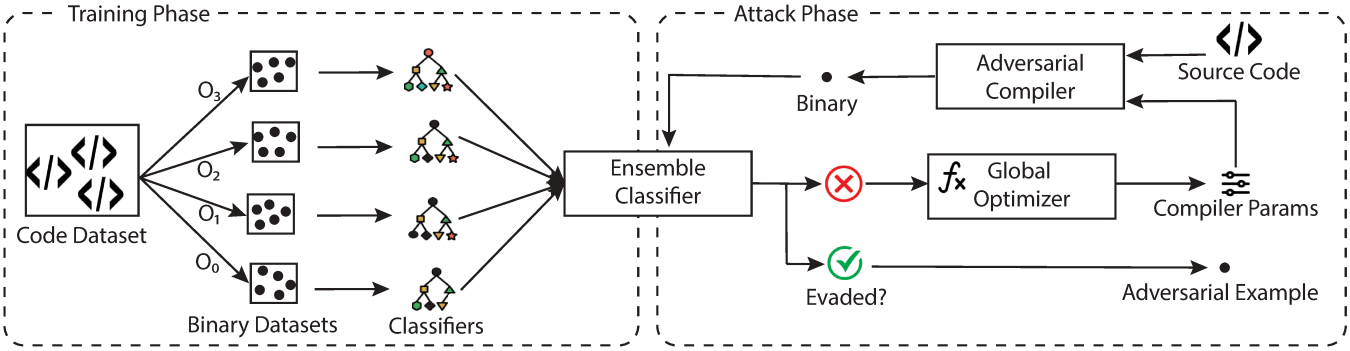
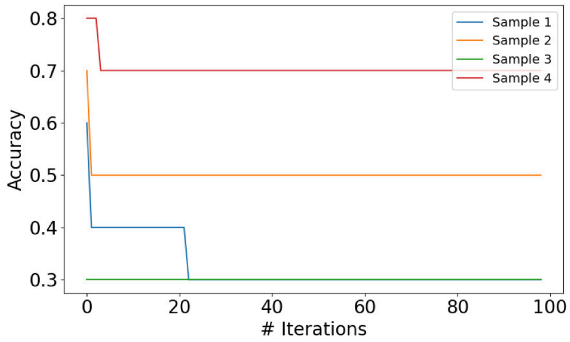
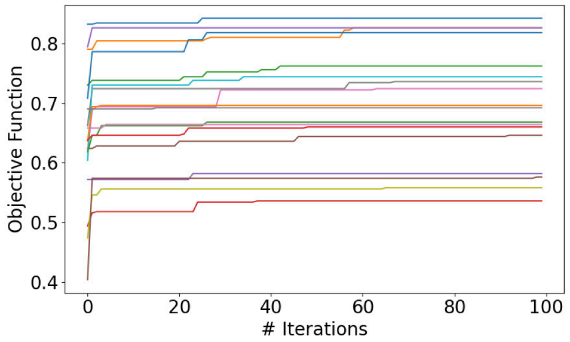


Figure 3: An overview of our attack methodology.



(a) Line graph depicting the drop in classifier accuracy as the number of iterations increases in each of our four experiments.



(b) Line graph depicting the improvements made to the objective function over time when an adversarial example was not found.

Figure 4: Summarizes our attack results.

6.2 Experimental Design

For each experiment, we randomly sample 10 authors with 8 programs per author from the Google Code Jam dataset. We use Caliskan *et al's* [9] model as an authorship attribution oracle, which our framework aims to evade. This model was chosen because it represents the state-of-the-art in binary stylometry. Note that, since, our framework does not directly depend on Caliskan*et al's* model [9], it could potentially be used to evade any such system.

To build the authorship attribution model for these 10 programmers, we use 7 programs for each. The remaining 10 programs are used to perform 10 attacks on the model, one for each 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. 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.

6.3 Replication of Caliskan *et al.*

The evaluation of our method critically depends on the performance of Caliskan et al's model [9]. For sanity checking, we replicated the work of Caliskan *et al.* [9]. 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.

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 [31]. 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.

6.4 Results

Finding adversarial examples. We have completed a total of 40 attacks, using 4 random and independent samples of 10 authors from the Google Code Jam Dataset. Our results are presented in Table 2. For each sample, we give the success rate, which is the proportion of programs in that sample for which we successfully found an adversarial example within 100 iterations. We also describe the average number of iterations required to find an adversarial example in successful attacks. Finally, we present the average value of the objective function attained in unsuccessful attacks (recall that our objective function is equal to 1 minus the confidence of the most confident correct attribution).

In all cases, the classifier models attained an accuracy of roughly 85% on the training dataset, with 7 instances per author. Our attack succeeded in reducing this accuracy to 45%. In cases where an

Sample	Success Rate	Successful Attack Duration	Failed Attack Avg. Value
1	0.7	4.43	0.81
2	0.5	1.4	0.69
3	0.7	1.0	0.71
4	0.3	2.0	0.65
Total	0.55	2.32	0.7

Table 2: Summarizes the results of our experiments. The success rate is the proportion of times our attack produced an adversarial example. Also presented is the average number of iterations required to find an adversarial example, and the average value of the objective function when an adversarial example could not be found after 100 iterations. Statistics are presented for each sample individually, as well as in aggregate.

adversarial example was found, the average number of iterations required was only 2.32. In other words, simply choosing a random selection of optimizations was often sufficient to produce an adversarial example. However, when this initial random selection failed, our technique typically failed to find an adversarial example within 100 iterations. Nonetheless, even when the attack failed, the attribution of the binary on average had only 30% confidence.

Impact of the number of iterations. We show the impact of the number of iterations to find adversarial examples in Figure 4. Figure (a) reinforces the observation that most adversarial examples were found in the first few iterations. However, from figure (b), we can see that the system continued to find modest improvements to the objective function throughout, decreasing the confidence of the final attribution.

The high success rate of our initial query supports our hypothesis that using atypical combinations of optimizations can substantially interfere with binary stylometry, even in the absence of repeated rounds of refinement. At the same time, however, repeated iterations led to only marginal improvement in the effectiveness of the attack. It is possible that this is a result of insufficient computational resources, or an indication that we chose too high a number for the dimensionality of the problem. But it may also indicate limitations on the possibility of crafting extremely effective adversarial examples using compiler optimizations alone – further data will be required to form more definitive judgments.

7 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 [9]. 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 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.

While our technique is non-deterministic, it does not make any strong guarantees about the distribution of optimizations it will ultimately choose. It is therefore conceivable that a sophisticated defender could utilize our technique to produce a representative set of optimizations and use them to train a classifier which would be robust against our attack. To combat this, the attacker could begin their attack by randomly choosing some percentage of the possible optimizations, and restricting their search only to those. By increasing or decreasing this percentage, the attacker would be able to make a tradeoff between the power and flexibility of the attack on the one hand, and the ease with which the defender could predict the optimizations at play on the other.

Finally, several interesting questions arose during the course of this work which could not be answered because of constraints on time and resources. 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.

8 RELATED WORK

8.1 Code Stylometry

Formal research into code stylometry began in the 1970s, where it was primarily focused on the problem of plagiarism detection [37] [17]. 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 [19, 48, 56].

Study of authorship attribution — that is, research premised on the idea that authors have unique and identifiable fingerprints — began in the late 80s [36], and was strongly influenced by the work of Spafford and Weeber[50] 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 [20, 21], which they showed to by 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 [4, 10, 41].

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 [10]. More recently, Abuhamad *et al.* report 92% accuracy in classifying 8903 authors, using primarily lexical features and recurrent neural networks [1].

8.2 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 [46]. Caliskan *et al.*, whose work

is the subject of the attack described in this paper, improved on Rosenblum's results through the use of a random forest classifier [9]. 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 [32].

8.3 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 [8]. Muir *et al.* investigated the possibility of using compiler optimization and static linkage to evade binary stylometry. They achieved modest decreases in accuracy, but only considered two different levels of optimization [34]. 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 [31]. 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 [42].

While not related to stylometry as such, Ren et al [45]. investigate the impact of non-standard optimizations on binary diffing. Using the genetic algorithm to select optimizations, they were able to create effective adversarial examples for malware detection tools.

There are several studies on evading textual stylometric classifiers [24, 28, 29, 44, 49]. In [44], authors proposed a round trip translation-based method to break stylometric linkability, but this was shown to be infefective in practice [30]. Later, McDonald *et al.* proposed a machine learning-based tool which would produce a sequence of suggestions enabling users to anonymize their own documents [30]. Shetty *et al.* proposed A^{4}NT [49], which used an approach similar to generative adversarial networks (GAN) [23] to anonymize documents while preserving semantics. A^{4}NT is suitable for mapping the writing style of a group of people to another group. Mahmood *et al.* proposed a semantic guided mutation-based approach to obfuscate stylistic features of a given text [29]. Recently, Gröndahl and Asokan [24] and Krishna *et al.* [28] have showed the feasibility of style transfer by leveraging recent advances in text paraphrasing.

9 CONCLUSION

In this paper, we showed that compiler optimization plays a bigger role in the accuracy of binary code stylometry than previously understood. Based on this insight, we developed an evasion technique to protect programmers' privacy against systematic surveillance. The experimental evaluation on the Google Code Jam dataset shows the effectiveness of our approach. Note that our method does not directly produce evading samples for the programmer. Instead, it recommends a set of compiler optimization flags for the compilation to produce an evading sample, providing more transparency than traditional transformation-oriented feature-perturbation-based evasion techniques.

ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation under grant no. 1908313.

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] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1985.

[3] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Gasol: Gas analysis and optimization for ethereum smart contracts. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 118–125, 2020.

[4] 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.

[5] Anish Athalye, Nicholas Carlini, and David A. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 274–283. PMLR, 2018.

[6] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[7] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.

[8] 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.

[9] 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. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS)*, February 2018.

[10] 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*, pages 255–270, 2015.

[11] David Callahan, Keith D Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. *ACM SIGPLAN Notices*, 21(7):152–161, 1986.

[12] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 39–57. IEEE Computer Society, 2017.

[13] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 442–446, 2017.

[14] Christian Collberg, Edward Carter, Saumya Debray, Andrew Huntwork, John Kececioglu, Cullen Linn, and Martin Stepp. Dynamic path-based software watermarking. In *Proc. ACM SIGPLAN '04 Conference on Programming Language Design and Implementation (PLDI-2004)*, June 2004.

[15] Hung Dang, Yue Huang, and Ee-Chien Chang. Evading classifiers by morphing in the dark. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 119–133. ACM, 2017.

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

[17] 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.

[18] Maciej Eder. Rolling stylometry. *Digital Scholarship in the Humanities*, 31(3):457–469, 2016.

[19] 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.

[20] 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.

[21] 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.

[22] Peter I Frazier. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[23] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 2672–2680, 2014.

[24] Tommi Gröndahl and N. Asokan. Effective writing style transfer via combinatorial paraphrasing. *Proc. Priv. Enhancing Technol.*, 2020(4):175–195, 2020.

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

[26] 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.

[27] Mahmut Kandemir, N Vijaykrishnan, and Mary Jane Irwin. Compiler optimizations for low power systems. In *Power aware computing*, pages 191–210. Springer, 2002.

[28] Kalpesh Krishna, John Wieting, and Mohit Iyyer. Reformulating unsupervised style transfer as paraphrase generation. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 737–762. Association for Computational Linguistics, 2020.

[29] Asad Mahmood, Faizan Ahmad, Zubair Shafiq, Padmini Srinivasan, and Fareed Zaffar. A girl has no name: Automated authorship obfuscation using mutant-x. *Proc. Priv. Enhancing Technol.*, 2019(4):54–71, 2019.

[30] Andrew W. E. McDonald, Sadia Afroz, Aylin Caliskan, Ariel Stolerman, and Rachel Greenstadt. Use fewer instances of the letter "i": Toward writing style anonymization. In *Privacy Enhancing Technologies - 12th International Symposium, PETS 2012, Vigo, Spain, July 11-13, 2012. Proceedings*, pages 299–318, 2012.

[31] Xiaozhu Meng, Barton P Miller, and Somesh Jha. Adversarial binaries for authorship identification. *arXiv preprint arXiv:1809.08316*, 2018.

[32] 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.

[33] 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.

[34] Macaully Muir and Johan Wikström. Anti-analysis techniques to weaken author classification accuracy in compiled executables, 2016.

[35] Luis Muñoz-González, Battista Biggio, Ambra Demontis, Andrea Paudice, Vasin Wongrassamee, Emil C. Lupu, and Fabio Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. In Bhavani M. Thuraisingham, Battista Biggio, David Mandell Freeman, Brad Miller, and Arunesh Sinha, editors, *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security, AISec@CCS 2017, Dallas, TX, USA, November 3, 2017*, pages 27–38. ACM, 2017.

[36] 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.

[37] Karl J Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM Sigcse Bulletin*, 8(4):30–41, 1976.

[38] James Pallister, Simon J Hollis, and Jeremy Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, 58(1):95–109, 2015.

[39] Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*.

[40] 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.

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

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

[43] 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.

[44] Josyula R. Rao and Pankaj Rohatgi. Can pseudonymity really guarantee privacy? In *9th USENIX Security Symposium, Denver, Colorado, USA, August 14-17, 2000*, 2000.

[45] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study. In *Proceedings of the ACM International Conference on Programming Language Design and Implementation (PLDI)*, June 2021.

[46] 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.

[47] 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.

[48] 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.

[49] Rakshith Shetty, Bernt Schiele, and Mario Fritz. A4NT: author attribute anonymity by adversarial training of neural machine translation. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1633–1650. USENIX Association, 2018.

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

[51] Nedim Srndic and Pavel Laskov. Practical evasion of a learning-based classifier: A case study. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 197–211. IEEE Computer Society, 2014.

[52] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[53] Florian Tramèr, Jens Behrmann, Nicholas Carlini, Nicolas Papernot, and Jörn-Henrik Jacobsen. Fundamental tradeoffs between invariance and sensitivity to adversarial perturbations. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 9561–9571. PMLR, 2020.

[54] Florian Tramèr, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. On adaptive attacks to adversarial example defenses. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[55] 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.

[56] Geoff Whale. Identification of program similarity in large populations. *The Computer Journal*, 33(2):140–146, 1990.